

征订版

Python

——语言基础与典型应用

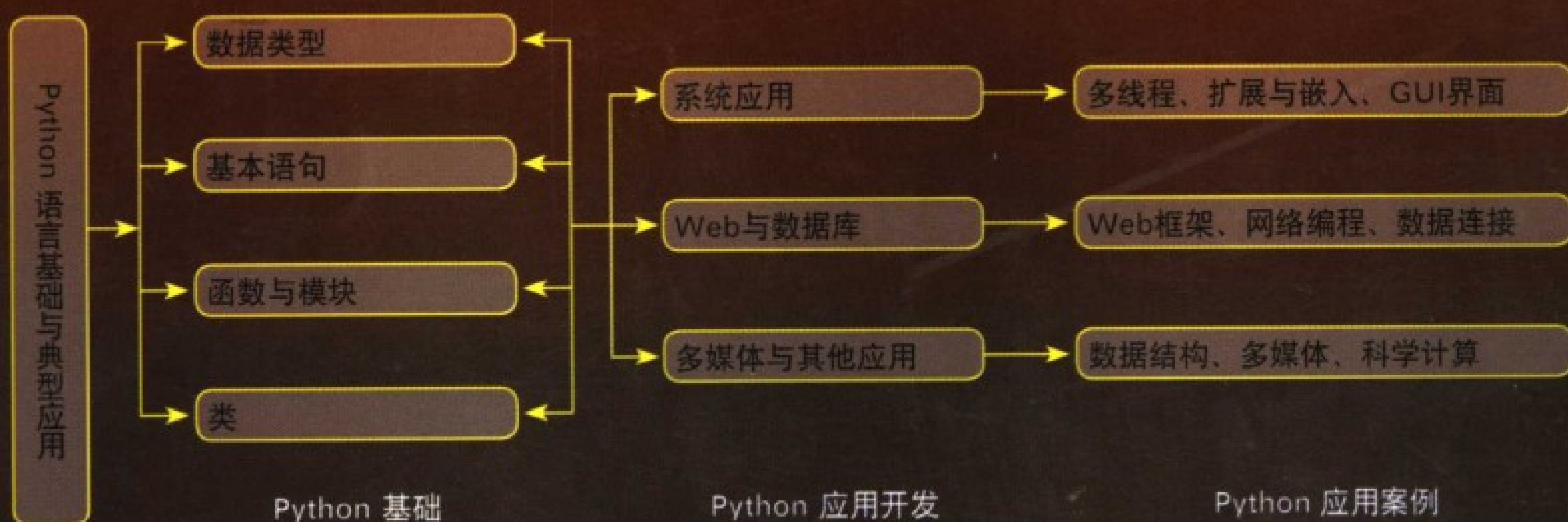
孙广磊 编著



 人民邮电出版社
POSTS & TELECOM PRESS

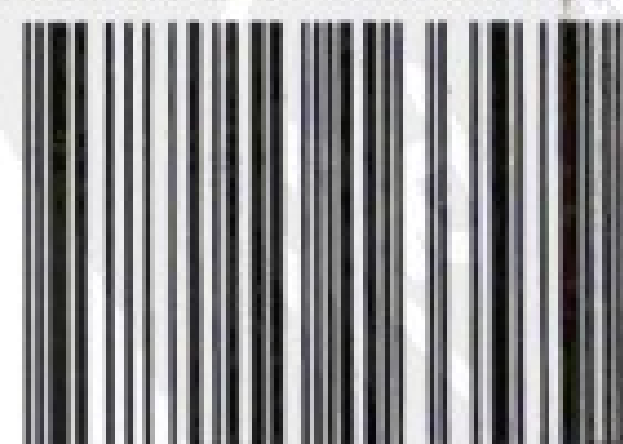
- 内容全面：涵盖了Web编程、数据库操作、图片处理等10大方面的Python应用
- 实用性强：给出151个应用实例，如批量文件重命名、生成缩略图、为图片添加水印等
- 内容新颖：对Python最新的Zope/Plone框架进行了讲解，介绍了如何使用Python进行Web开发
- 深入系统：给出了Python在Windows下与系统相关的操作，如查看修改系统启动项等

Python 基础与典型应用学习路线



分类建议：计算机 / 网络技术
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-16657-9



9 787115 166579 >

ISBN 978-7-115-16657-9/TP

定价：59.00 元（附光盘）

TP312/2526D

2007

征订

Python

——语言基础与典型应用

孙广磊 编著

人民邮电出版社
北京

图书在版编目 (CIP) 数据

征服 Python: 语言基础与典型应用 / 孙广磊编著. —北京: 人民邮电出版社, 2007.9
ISBN 978-7-115-16657-9

I. 征... II. 孙... III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字 (2007) 第 122531 号

内 容 提 要

Python 是目前流行的脚本语言之一。本书由浅入深、循序渐进地讲解如何使用 Python 进行程序开发。全书内容包括 Python 安装、开发工具简介、Python 基本语法、系统应用、GUI 编程、数据库和网络编程、数据结构与算法、多媒体编程、图片处理等。书中针对 Python 的扩展模块给出了详细的语法介绍, 并且提供了典型实例, 使读者能很快地使用 Python 进行程序开发。

本书适合 Python 初学者、程序设计人员、编程爱好者、大专院校学生以及需要进行科学计算的工程人员阅读。

征服 Python——语言基础与典型应用

- ◆ 编 著 孙广磊
责任编辑 屈艳莲
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
- ◆ 开本: 800×1000 1/16
印张: 31
字数: 672 千字
印数: 1—5 000 册
- 2007 年 9 月第 1 版
2007 年 9 月北京第 1 次印刷

ISBN 978-7-115-16657-9/TP

定价: 59.00 元 (附光盘)

读者服务热线: (010)67132692 印装质量热线: (010)67129223



前 言

Python 是一种功能强大的脚本语言，使用 Python 可以完成从文本处理到创建复杂的 3D 图形等各种工作。在企业级应用中，由于 Python 具有简洁的语法、丰富的扩展模块，使用它可以大幅缩短开发周期，节约成本。

另外，JPython 还可以在 Java 中使用 Python，通过 Python 的灵活性来提高 Java 在企业级应用中的效率。在 Web 方面，有很多基于 Python 的流行 Web 框架，如 Zope、Plone、Django、TurboGears，通过这些 Web 框架，程序员可以使用 Python 迅速地构建安全、功能强大的网站。

在数值计算与工程应用中，Python 与传统的 C 和 Fortran 相比，更加灵活、简洁，并且可以十分方便地创建 GUI 界面。通过使用 SciPy 模块和 Matplotlib 模块，可以进行数值计算、实现工程数据的可视化。

本书特色

- (1) 内容全面：涵盖了 Python 应用的各个方面，如 Web 编程、数据库操作、图片处理、科学计算等。
- (2) 内容翔实：书中不仅对模块函数进行了详细的介绍，而且给出了应用实例。
- (3) 实用性强：书中给出一些应用实例，可以完成日常工作中许多繁琐的操作，如批量文件重命名、修改图片大小等。
- (4) 内容新颖：对 Python 最新的 Web 框架应用进行了讲解。
- (5) 实例典型：突出 Python 的实用性，如使用 Python 连接 Access 数据库、连接 Gtalk、查看天气预报等。
- (6) 实例丰富：针对 Python 的应用都给出了详细的实例。
- (7) 与系统结合紧密：给出了 Python 在 Windows 下与系统相关的操作。

本书的内容

本书分为 5 篇，分别是：Python 入门、Python 语法、系统应用、Web 与数据库以及多

媒体与其他应用。其中，Python 入门篇包括第 1 章和第 2 章，主要是 Python 的基础部分。Python 语法篇包括第 3 章至第 7 章，主要介绍 Python 的语法。系统应用篇包括第 8 章至第 15 章，主要介绍了 Python 的系统编程和 GUI 编程。Web 与数据库篇包括第 16 章至第 19 章，主要介绍了 Python 的网络应用、数据库应用以及 HTML 和 XML 的处理。多媒体与其他应用篇包括第 20 章至第 23 章，主要介绍了 Python 的多媒体编程、数据结构与算法、科学计算和图片处理。

第 1 章介绍了 Python 的发展历史、衍生版本、开发环境的搭建以及脚本的运行，并且创建了第一个 Python 脚本。

第 2 章介绍了 Python 脚本的结构、基本输入/输出、在 Python 中使用中文以及 Python 中的数学运算。本章主要介绍了 Python 中一些比较零散，但又经常困扰初学者的问题。

第 3 章介绍了 Python 的数据类型与基本语句，包括数字、字符串、列表、元组、字典、文件以及控制语句。

第 4 章介绍了 Python 的函数与模块，包括函数的声明与调用、参数的传递、参数的作用域、lambda 表达式以及模块的创建和使用。

第 5 章介绍了正则表达式，包括 re 模块的函数、对象和方法，以及如何使用正则表达式进行文本处理。

第 6 章介绍了使用 Python 进行面向对象的编程，包括如何在 Python 中定义类、如何定义类的属性和方法、类的继承、方法的重载以及如何在模块中包含类。

第 7 章介绍了 Python 的异常处理和调试，包括捕获异常、引发异常、使用 pdb 模块调试脚本以及在 PythonWin 中调试脚本。

第 8 章介绍了 Python 的扩展和嵌入。通过编写 Python 扩展，可以增加 Python 的功能，通过在 C/C++ 编写的应用程序中嵌入 Python，可以使用 Python 的强大功能。

第 9 章介绍了 Python 的多线程编程，包括线程基础、线程同步、线程间通信以及 Python 的修改版 Stackless Python 中的微线程。

第 10 章介绍了 Python 脚本在 Windows 系统下的应用，包括访问注册表、处理目录和文件、生成可执行文件以及运行其他程序。

第 11 章介绍了使用 PythonWin 编写图形用户界面的方法，包括使用 PythonWin 创建窗口、对话框和菜单。

第 12 章介绍了使用 Tkinter 编写图形用户界面的方法，包括使用 Tkinter 创建窗口、创建组件、事件处理以及创建对话框。

第 13 章介绍了使用 wxPython 编写图形用户界面的方法，包括使用 wxPython 创建窗口、组件、对话框、菜单以及资源文件的创建和使用。另外，本章最后还给出一个使用 wxPython 创建简单的记事本的例子。

第 14 章介绍了使用 PyGTK 编写图形用户界面的方法，包括使用 PyGTK 创建窗口、组件、对话框、菜单以及资源文件的创建和使用。

第 15 章介绍了使用 PyQt 编写图形用户界面的方法，包括使用 PyQt 创建窗口、组件、对话框、菜单以及资源文件的创建和使用。

第 16 章介绍了 Python 与数据库的连接，包括使用 ODBC、DAO、ADO 连接 Access 数据库，使用 MySQLdb 模块连接 MySQL 数据库以及在 Python 中使用小巧的 SQLite 数据库。

第 17 章介绍了 Python 的 Web 应用，包括 Zope 的安装和使用、Plone 的安装和使用、在 IIS 中使用 Python 创建网站以及在 Apache 中使用 Python 创建网站。

第 18 章介绍了 Python 的网络编程，包括低级 socket 的使用、局域网中文件的传输、网站的访问、FTP 的访问、邮件的收取和发送以及使用 Python 连接到 Gtalk。

第 19 章介绍了使用 Python 处理 HTML 和 XML 的方法，包括 HTML 的分析、XML 的基础以及在 Python 中处理 XML 模块。另外，本章最后还使用 Python 创建一个简单的 RSS 阅读器。

第 20 章介绍了基本的数据结构和算法，包括表、栈、队列、树的创建和遍历、图的创建和遍历以及基本的查找算法和排序算法。

第 21 章介绍了使用 Python 进行科学计算的方法，包括 NumPy 和 SciPy 的安装和使用、矩阵运算、线性方程组的求解以及使用 Matplotlib 函数绘制图形。

第 22 章介绍了使用 Python 进行多媒体编程的方法，包括使用 PyOpenGL 创建 3D 图形、播放音频文件以及游戏编程等。

第 23 章介绍了使用 Python 处理图片的方法，包括 PIL 的安装、图片文件格式的转换、缩略图的生成以及为图片添加 Logo。

如何学习本书

Python 语法简单，应用灵活。本书的前两篇主要介绍 Python 的基础语法，通过对本书前两篇的学习，读者可以使用 Python 编写简单的脚本。读者应该认真阅读前两篇，为后边的学习打好基础，其中第二篇中第 5 章的正则表达式难度较大，读者可以通过大量的练习慢慢掌握。

本书的第三篇至第五篇主要是 Python 的实际应用，读者可以根据需要有选择地进行学习。这几篇中的内容较前两篇难度大，读者需要在掌握 Python 的基本语法之后进行学习，其中第 8 章需要读者有 C/C++ 的相关知识，如果读者不会使用 C/C++ 进行编程，可以跳过该章。另外，书中对 Python 的几种 GUI 编程工具都进行了讲解，读者可以比较各种 GUI 编程工具，根据自己的兴趣选择适合自己使用的 GUI 编程工具。

书中第 20 章介绍了基本的数据结构和算法，通过对第 20 章的学习，读者可以更深刻地

认识程序设计，便于读者更进一步学习。书中的第 21 章介绍了使用 Python 进行科学计算，结合 Python 的 GUI 编程以及数据结构和算法，工程人员可以使用 Python 快速编写用于工程设计、计算的脚本工具。

适合的读者

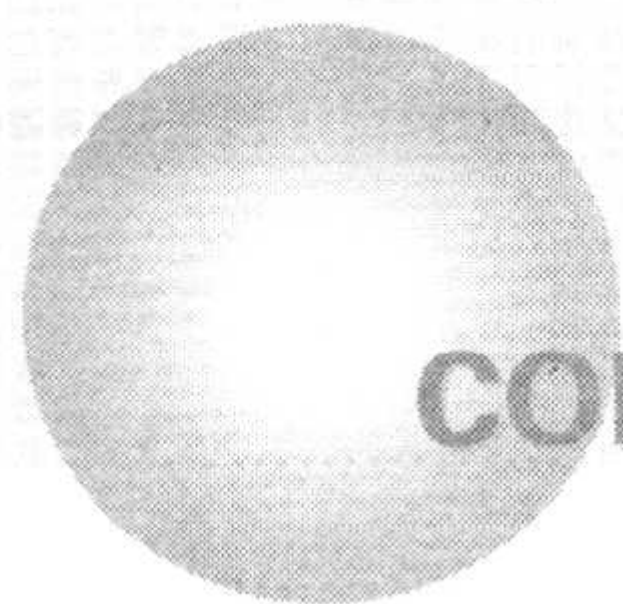
- Python 初学者；
- 程序设计人员；
- 编程爱好者；
- 大专院校学生；
- 需要进行科学计算的工程人员。

参与本书编写的人员

本书由孙广磊统筹编写，同时参与编写的还有刘丹、陈冠军、罗思红、孙飞、王朋章、王石磊、王新平、文奇、吴琪、席国庆、谢超文、臧勇、张国强、张家春、郭玉敏、贺道权、胡斯登、江成海、姜海峰、李峥、利建昌、陈杰、刘波等，在此一并表示感谢。

编 者

2007 年 8 月



CONTENTS

目 录

第一篇 Python 入门

第 1 章 Python 概述	3
1.1 Python 简介	3
1.2 为什么使用 Python	4
1.3 不同平台下的 Python	5
1.3.1 Java 平台下的 Python	5
1.3.2 .NET 平台下的 Python: Python for .NET 和 IronPython	6
1.4 搭建开发环境	6
1.4.1 对操作系统的要求	6
1.4.2 下载和安装 Python	7
1.4.3 自己编译 Python	9
1.4.4 使用 Vim 编写 Python 脚本	10
1.4.5 使用 Emacs 编写 Python 脚本	14
1.4.6 使用 PythonWin 编写 Python 脚本	16
1.4.7 其他的 Python 开发环境	18
1.5 运行 Python 脚本	19
1.5.1 第一个 Python 程序——“Hello, Python!”	19
1.5.2 在 Python 交互式命令行中运行脚本	20
第 2 章 Python 起步	22
2.1 脚本基本结构	22
2.2 基本输入/输出	24
2.3 在 Python 中使用中文	26
2.4 把 Python 当作计算器	28

第二篇 Python 语法

第 3 章 Python 数据类型与基本语句	33
------------------------------	----

3.1	Python 数据类型——数字	33
3.1.1	基本类型	33
3.1.2	运算符	34
3.2	Python 数据类型——字符串	35
3.2.1	字符串概述	36
3.2.2	操作字符串	36
3.2.3	索引和分片	39
3.2.4	格式化字符串	40
3.2.5	字符串与数字相互转换	40
3.2.6	原始字符串 (Raw String)	41
3.3	Python 数据类型——列表和元组	41
3.4	Python 数据类型——字典	42
3.5	Python 数据类型——文件	43
3.6	Python 基本语句	45
3.6.1	if 语句	45
3.6.2	for 语句	47
3.6.3	while 语句	49
第 4 章	函数与模块	51
4.1	函数	51
4.1.1	函数声明	51
4.1.2	函数调用	52
4.2	函数中的参数	53
4.2.1	参数默认值	53
4.2.2	参数传递	54
4.2.3	可变长参数	55
4.2.4	参数引用	56
4.3	作用域	56
4.4	lambda 表达式	57
4.5	模块	58
4.5.1	模块概述	58
4.5.2	模块查找路径	60
4.5.3	模块编译	61
4.5.4	模块独立运行——_name_属性	62
4.5.5	dir()函数	62

4.6	模块包.....	63
第 5 章	正则表达式	65
5.1	正则表达式概述	65
5.1.1	基本元字符	65
5.1.2	常用正则表达式分析	66
5.2	re 模块函数应用	67
5.2.1	匹配和搜索	67
5.2.2	替换函数	68
5.2.3	分割字符串函数	69
5.3	正则表达式对象	69
5.3.1	以 “\” 开头的元字符	69
5.3.2	编译正则表达式	70
5.3.3	使用原始字符串	71
5.4	正则表达式对象的属性和方法	71
5.4.1	匹配和搜索	71
5.4.2	替换	73
5.4.3	分割字符串	74
5.5	使用组	75
5.5.1	组概述	75
5.5.2	组的扩展语法	76
5.6	Match 对象	77
5.6.1	使用 Match 对象处理组	77
5.6.2	使用 Match 对象处理索引	78
5.7	使用正则表达式处理文件	79
第 6 章	面向对象的 Python	82
6.1	概述	82
6.1.1	Python 中的面向对象的思想	82
6.1.2	类和对象	83
6.2	类的基础	84
6.2.1	类的定义	84
6.2.2	类的使用	85
6.3	类的属性和方法	86
6.3.1	类的属性	86
6.3.2	类的方法	87

6.4	类的继承.....	90
6.4.1	通过继承创建类	90
6.4.2	多重继承	91
6.5	重载	93
6.5.1	方法重载	93
6.5.2	运算符重载	94
6.6	模块中的类.....	96
第 7 章	异常与调试.....	98
7.1	捕获异常.....	98
7.1.1	使用 try 语句	98
7.1.2	处理异常	100
7.1.3	多重异常处理	102
7.2	引发异常.....	103
7.2.1	使用 raise 引发异常	103
7.2.2	assert——简化的 raise 语句	104
7.2.3	自定义异常类	105
7.3	使用 pdb 调试 Python 脚本	105
7.3.1	运行语句	105
7.3.2	运行表达式	106
7.3.3	运行函数	107
7.3.4	设置硬断点	107
7.3.5	pdb 调试命令	108
7.4	在 PythonWin 中调试 Python 脚本	110

第三篇 系统应用

第 8 章	Python 扩展和嵌入	117
8.1	扩展 Python	117
8.1.1	扩展概述	117
8.1.2	程序详解	122
8.1.3	在 Python 扩展中使用 MFC	124
8.2	在 C/C++ 中嵌入 Python	128
8.2.1	高层次嵌入 Python	128
8.2.2	较低层次嵌入 Python	129
8.2.3	在 C 中嵌入 Python 实例	133

8.3	语言的黏合剂 SWIG	135
8.3.1	在 Windows 集成开发环境中使用 SWIG.....	136
8.3.2	SWIG 接口文件的语法简介	138
8.4	混合系统接口 Boost.Python	139
8.4.1	编译 Boost.Python.....	139
8.4.2	使用 Boost.Python 扩展和嵌入 Python.....	140
8.4.3	使用 Pyste 代码生成器	144
8.5	连接 Python 与 C 的桥梁——Pyrex	145
8.5.1	安装使用 Pyrex	145
8.5.2	Pyrex 文件语法	146
第 9 章	多线程编程	148
9.1	线程基础.....	148
9.1.1	创建线程	148
9.1.2	Thread 对象中的方法.....	150
9.2	线程同步.....	153
9.2.1	简单的线程同步.....	153
9.2.2	使用条件变量保持线程同步	154
9.2.3	使用队列保持线程同步	156
9.3	线程间通信	157
9.3.1	Event 对象的方法	157
9.3.2	使用 Event 对象实现线程间通信	158
9.4	微线程——Stackless Python.....	158
9.4.1	Stackless Python 概述.....	159
9.4.2	使用微线程	161
第 10 章	系统编程.....	162
10.1	访问 Windows 注册表.....	162
10.1.1	注册表概述	162
10.1.2	使用 Python 操作注册表	163
10.1.3	查看系统启动项.....	166
10.1.4	修改 IE	167
10.2	文件和目录	169
10.2.1	文件目录常用函数.....	169
10.2.2	批量重命名	171
10.2.3	代码框架生成器.....	172

10.3	使用 py2exe 生成可执行文件	173
10.3.1	安装 py2exe.....	173
10.3.2	使用 py2exe 生成可执行文件	174
10.4	运行其他程序	176
10.4.1	使用 os.system 函数运行其他程序	176
10.4.2	使用 ShellExecute 函数运行其他程序	176
10.4.3	使用 CreateProcess 函数运行其他程序	177
10.4.4	使用 ctypes 调用 kernel32.dll 中的函数	178
第 11 章	使用 PythonWin 编写 GUI	184
11.1	Windows GUI 编程概述	184
11.1.1	使用 Windows API 创建窗口	184
11.1.2	使用 MFC 创建窗口	186
11.2	对话框	188
11.2.1	创建对话框	188
11.2.2	向对话框中添加控件	188
11.2.3	使用 DLL 文件中的资源	191
11.2.4	处理按钮消息	192
11.3	菜单	195
11.3.1	创建菜单	195
11.3.2	使用 DLL 中的菜单	198
11.3.3	处理菜单消息	200
第 12 章	使用 Tkinter 编写 GUI	202
12.1	Tkinter 概述	202
12.1.1	创建简单的窗口	202
12.1.2	向窗口中添加组件	203
12.2	使用组件	204
12.2.1	组件分类	204
12.2.2	组件布局	204
12.2.3	使用按钮	205
12.2.4	使用文本框	207
12.2.5	使用标签	208
12.2.6	使用菜单	210
12.2.7	使用单选框和复选框	211
12.2.8	绘制图形	214

12.3	事件处理	216
12.3.1	事件表示	216
12.3.2	响应事件	218
12.4	创建对话框	221
12.4.1	使用标准对话框	221
12.4.2	创建自定义对话框	226
第 13 章	使用 wxPython 编写 GUI	228
13.1	wxPython 概述	228
13.1.1	安装 wxPython	228
13.1.2	创建窗口	230
13.2	组件	231
13.2.1	面板	231
13.2.2	按钮	233
13.2.3	标签	235
13.2.4	文本框	236
13.2.5	单选框和复选框	239
13.2.6	使用 sizer 布置组件	241
13.3	对话框	242
13.3.1	消息框和标准对话框	242
13.3.2	创建自定义对话框	245
13.4	菜单	246
13.4.1	创建菜单	246
13.4.2	绑定菜单事件	248
13.5	资源文件	249
13.5.1	创建资源文件	249
13.5.2	在脚本中使用资源文件	251
13.6	一个简单的文本编辑器	252
第 14 章	使用 PyGTK 编写 GUI	256
14.1	PyGTK 概述	256
14.1.1	PyGTK 安装	256
14.1.2	创建窗口	257
14.2	组件	258
14.2.1	标签	258
14.2.2	按钮	261

14.2.3	容器组件	264
14.2.4	文本框	266
14.2.5	单选框和复选框	269
14.3	消息框和对话框	271
14.3.1	消息框	271
14.3.2	标准对话框	273
14.3.3	自定义对话框	275
14.4	菜单	277
14.4.1	创建菜单	277
14.4.2	菜单事件	281
14.5	资源文件	282
14.5.1	使用 Glade 创建资源文件	282
14.5.2	使用资源文件	284
第 15 章	使用 PyQt 编写 GUI	286
15.1	PyQt 概述	286
15.1.1	PyQt 的安装	286
15.1.2	使用 PyQt 创建窗口	288
15.2	组件	288
15.2.1	标签	288
15.2.2	布局组件和空白项	289
15.2.3	按钮	292
15.2.4	文本框	294
15.2.5	单选框和复选框	297
15.2.6	菜单	298
15.3	对话框	301
15.3.1	消息框和标准对话框	301
15.3.2	自定义对话框	306
15.4	资源文件	307
15.4.1	使用 Qt Designer 创建资源文件	307
15.4.2	使用资源文件	309

第四篇 Web 与数据库

第 16 章	Python 与数据库	313
16.1	连接 Access 数据库	313

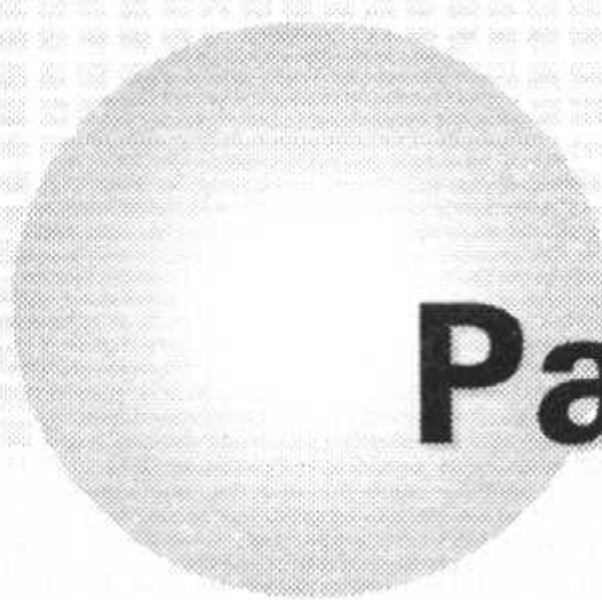
16.1.1	使用 ODBC 连接 Access 数据库	313
16.1.2	使用 DAO 连接 Access 数据库	317
16.1.3	使用 ADO 连接 Access 数据库	318
16.2	使用 MySQL 数据库	319
16.2.1	安装 MySQL	319
16.2.2	连接到 MySQL	322
16.3	嵌入式数据库 SQLite	324
第 17 章	Python Web 应用	326
17.1	开源 Web 应用服务器 Zope	326
17.1.1	安装 Zope	326
17.1.2	使用 Zope 管理界面	328
17.1.3	创建模板	331
17.1.4	添加 Python 脚本	334
17.1.5	连接 MySQL 数据库	335
17.2	使用 Plone 内容管理系统	337
17.2.1	安装 Plone	337
17.2.2	安装 Plone 产品	340
17.3	在 Microsoft IIS 中使用 Python	344
17.3.1	安装 Microsoft IIS	344
17.3.2	在 ASP 中使用 Python 脚本	347
17.3.3	一个简单的例子	350
17.4	在 Apache 中使用 Python	353
17.4.1	安装配置 Apache	353
17.4.2	安装 mod_python	355
17.4.3	使用 Python Sever Pages 创建留言板	357
第 18 章	Python 网络编程	361
18.1	使用 socket 模块	361
18.1.1	网络编程概述	361
18.1.2	使用 socket 模块建立网络通信	362
18.1.3	在局域网中传输文件	367
18.2	使用 urllib、httplib 以及 ftplib	370
18.2.1	使用 Python 访问网站	370
18.2.2	访问 FTP	374
18.3	使用 poplib 和 smtpplib 模块收发邮件	378

18.3.1	检查 E-mail	378
18.3.2	发送 E-mail	381
18.4	连接到 Gtalk	383
18.4.1	安装 XMPPY	384
18.4.2	使用 XMPPY	385
第 19 章	处理 HTML 与 XML	388
19.1	处理 HTML	388
19.1.1	HTMLParser 模块简介	388
19.1.2	获取页面图片地址	390
19.1.3	查看天气预报	391
19.2	处理 XML	397
19.2.1	XML 基础	397
19.2.2	文档类型定义	399
19.2.3	命名空间	400
19.3	使用 Python 处理 XML	401
19.3.1	使用 xml.parsers.expat 处理 XML	401
19.3.2	使用 xml.sax 处理 XML	404
19.3.3	使用 xml.dom 处理 XML	405
19.4	简单的 RSS 阅读器	405

第五篇 多媒体与其他应用

第 20 章	数据结构与算法	413
20.1	表、栈和队列	413
20.1.1	表	413
20.1.2	栈	414
20.1.3	队列	416
20.2	树和图	418
20.2.1	树	418
20.2.2	二叉树	419
20.2.3	图	422
20.3	查找与排序	424
20.3.1	查找	424
20.3.2	排序	426
第 21 章	科学计算	429

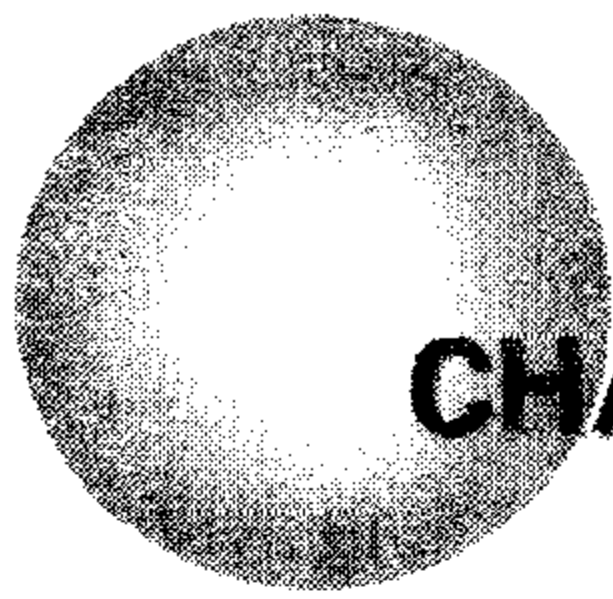
21.1 NumPy 和 SciPy 简介	429
21.1.1 安装 NumPy 和 SciPy	429
21.1.2 NumPy 简介	431
21.1.3 SciPy 简介	432
21.2 矩阵运算和解线性方程组	433
21.2.1 矩阵运算	434
21.2.2 解线性方程组	436
21.3 使用 Matplotlib 绘制函数图形	437
21.3.1 安装 Matplotlib	437
21.3.2 使用 Matplotlib 绘制图形	440
第 22 章 Python 多媒体编程	442
22.1 使用 PyOpenGL 绘制 3D 图形	442
22.1.1 安装 PyOpenGL	442
22.1.2 使用 PyOpenGL 创建窗口	443
22.1.3 绘制文字	444
22.1.4 绘制 2D 图形	446
22.1.5 绘制 3D 图形	448
22.1.6 纹理映射	450
22.2 播放音频文件	453
22.2.1 使用 DirectSound	453
22.2.2 使用 WMPPlayer.OCX	455
22.3 PyGame	456
22.3.1 安装 PyGame	457
22.3.2 使用 PyGame 编写简单的游戏	458
第 23 章 使用 PIL 处理图片	462
23.1 PIL 概述	462
23.1.1 安装 PIL	462
23.1.2 PIL 简介	463
23.2 使用 PIL 处理图片	465
23.2.1 转换图片格式	465
23.2.2 生成缩略图	467
23.2.3 为图片添加 Logo	471



Part 1

第一篇

Python 入门



CHAPTER 1

第 1 章 Python 概述

Python 是免费的解释性语言，具有面向对象的特性，可以运行在多种操作系统之上。Python 具有清晰的结构、简洁的语法以及强大的功能。Python 可以完成从文本处理到网络通信等各种工作。Python 自身已经提供了大量的模块来实现各种功能，除此以外还可以使用 C/C++ 来扩展 Python，甚至还可以将 Python 嵌入到其他语言中。

1.1 Python 简介

Python 是目前流行的脚本语言之一，它是由 Guido van Rossum 创建的。Python 语言主要受到教学语言 ABC 和 Modula-3 的影响，因此被设计得简洁、优美，却又不失脚本的灵活性和强大的功能。Python 的主要特点可以用图 1-1 来表示。

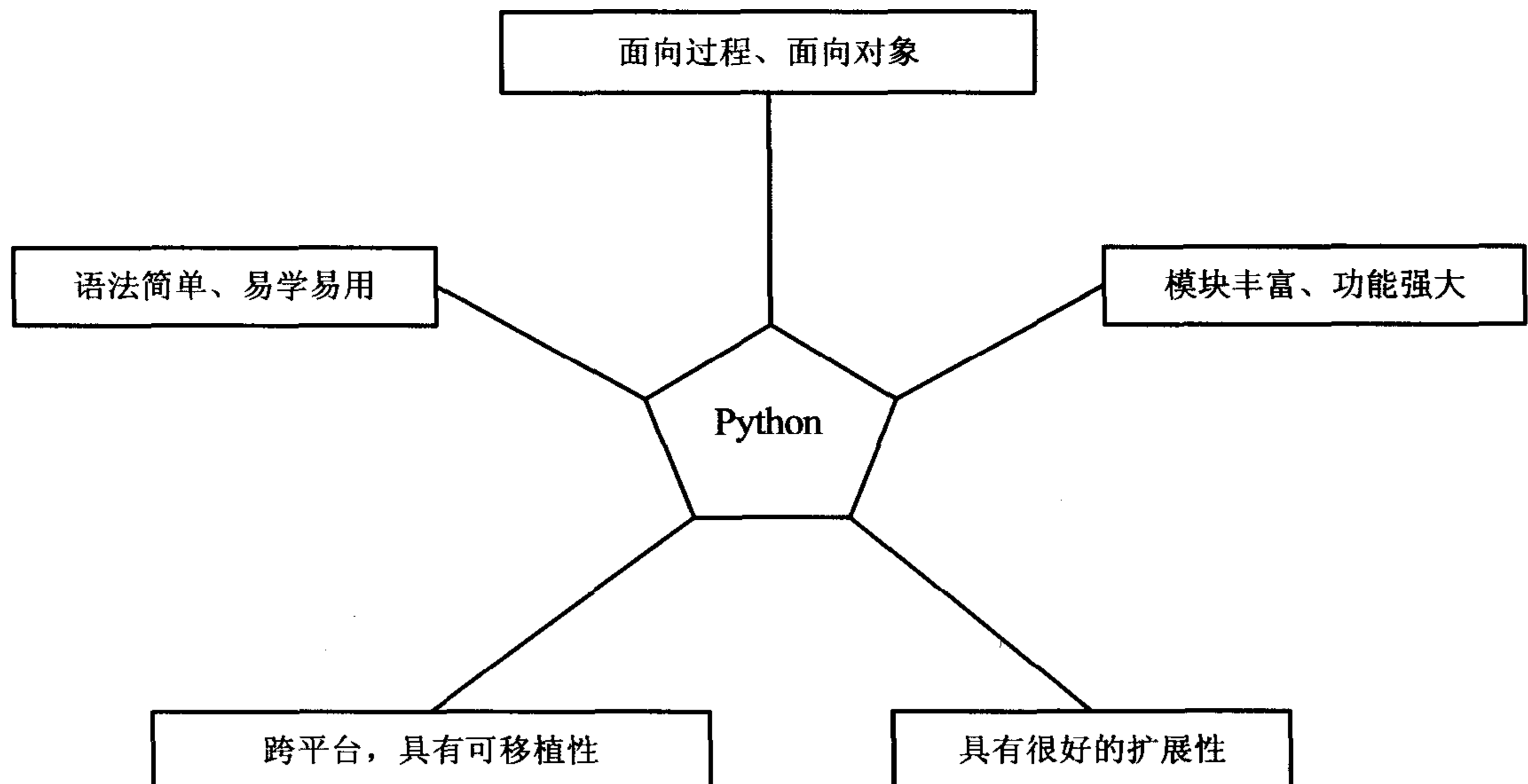


图 1-1 Python 特点

Python 最大的特点是其独特而又简单的语法。在 Python 中，可以以不同的缩进量来表示代码所属的语句块，也不需要使用 C/C++ 中的花括号“{}”。对于学过 C/C++ 并长期使用 C/C++ 的用户来说，可能对 Python 的缩进语法感到不适应，但强制性的缩进使得代码看起来很清晰，并且使用缩进在一定程度上也减少了代码输入量，因为省去了一对花括号。当然，前提是所使用的代码编辑器支持自动缩进。

Python 的语法很简单，对于初学者来说，学习并使用 Python 并不困难，甚至仅花费几天时间进行短暂的学习后就可以使用 Python 编写出“引以为傲”的非常实用的脚本。作为脚本语言的 Python 非常灵活，使用它可以实现各种各样的功能。虽然 Python 被称为脚本语言，通常来说脚本语言都是解释性的语言，不需要编译过程，但 Python 具有编译过程，它会将脚本编译成字节码的形式。一般不需要将脚本自己进行编译，Python 自己会根据需要编译。通常情况下，只有作为模块的脚本才会被编译成字节码的形式。

Python 遵循 GPL 协议，是源代码开放的软件。用户不仅可以免费使用 Python 来编写脚本，还可以阅读 Python 的源代码，了解 Python 的内部功能，甚至还可以参与到 Python 的开发之中，为 Python 的发展做出贡献。

1.2 为什么使用 Python

Python 不是唯一的选择，但 Python 却是不错的选择。为什么使用 Python，这首先取决于用户。面对简单易学而且功能强大的 Python，越来越多的人选择使用它来完成各种各样的任务。

1. Python 是自由软件

Python 遵循 GPL 协议，是自由软件。这是 Python 流行的原因之一。用户使用 Python 不需要支付任何费用，也不用担心版权的问题。Python 甚至可以用作商业用途，而且很多商业软件公司也开始将自己的产品变为开源的，例如 Java。作为开源软件的 Python 将具有更强的生命力。

作为自由软件，最令人鼓舞的就是可以阅读其源代码，发现其中的神奇之处。当深入地使用 Python 以后，可能会发现 Python 的某些特性，而这些特性并没有详细的文档说明，此时可以阅读 Python 的源代码去详细地了解 Python 的这些特性。

2. Python 是跨平台的

跨平台、良好的可移植性是 C 语言成为经典编程语言的关键，而 Python 正是由可移植的 ANSI C 编写的，这意味着在 Windows 下编写的 Python 脚本可以轻易地运行在 Linux 下。当然如果在 Python 脚本中使用了 Windows 的某些特性，比如 COM，那就另当别论了。Python 最早就是在 Mac 操作系统下实现的，有很强的可移植性；因此，如果将可移植性作为选择编程语言的首要考虑因素，Python 是很好的选择。

3. Python 功能强大

Python 强大的功能也许是很多用户支持 Python 的最重要原因。从字符串处理到复杂的

3D 编程, Python 借助扩展模块都可以轻易完成。实际上 Python 的核心模块已经提供了足够强大的功能, 使用它精心设计的内置对象可以完成功能强大的操作。

Python 可以使用在多个领域, 如系统编程, 帮助用户完成繁琐的日常工作; 科学计算, 它简洁的语法可以像使用计算器一样来完成科学计算; 快速原型, 它省去了编译调试的过程, 可以快速地实现系统原型; Web 编程, 使用它可以编写 CGI, 而现在流行的 Web 框架也可以使用 Python 实现。

4. Python 是可扩展的

(1) Python 提供了扩展接口, 通过使用 C/C++ 可以对 Python 进行扩展。

(2) Python 可以嵌入到 C/C++ 编写的程序之中。在 C/C++ 编写的程序中使用它可以完成一些对于 C/C++ 实现起来较复杂的任务。在某些情况下, 它可以作为动态链接库的替代品在 C/C++ 中使用。

(3) Python 可以很容易地被修改、调试, 而不需要重新编译。

5. Python 易学易用

(1) Python 的语法十分简单, 而且其中的数据类型的概念十分模糊。

(2) 在使用变量时无需事先声明变量的类型。

(3) 使用 Python 不必关心内存的使用, 它会自动地分配、回收内存。

(4) Python 提供了功能强大的内置对象和方法。

(5) 使用 Python 可以减少其他编程语言的复杂性, 例如在 C 语言中使用数十行代码实现的排序, 而在 Python 中, 可以使用列表的排序函数就可以轻易完成。

1.3 不同平台下的 Python

1.3.1 Java 平台下的 Python

Java 是一种面向对象的程序设计语言, 它基本上是仿照 C++ 进行开发的。与 C++ 不同的是, Java 是完全面向对象的, Java 舍弃了 C++ 中容易导致问题的指针。另外, Java 提供了自动垃圾回收的功能, 用于回收不再被使用的内存。Java 和 Python 一样, 也是跨平台的解释性的语言, 而且它们都能将代码编译。但相对于 Python 而言, Java 则过于庞大和复杂。

JPython 是由 Jim Hugunin 使用 Java 对“原始”Python 的重写。由于 JPython 的出现, 通常所说的 Python 也被称为 CPython, 因为其是由 C 编写的。CPython 是相对于 JPython 而言的。JPython 的出现使得 Python 可以在 Java 环境中运行。JPython 和 CPython 的运行方式是一样的, 在 JPython 下编写 Python 脚本和在 CPython 下编写 Python 脚本并没有太多的区别。

相对于 CPython 而言, JPython 要慢一些。这是由于 Java 是解释性的语言, JPython 本身

首先要由 Java 解释器进行解释，而编写的 Python 脚本又需要由 JPython 解释器进行解释，这当然要比 CPython 慢一些。多数情况下，这并不是什么大问题。由于 Java 的流行，有大量的 Java 可以使用。有了 Jpython，就可以在 Python 中非常方便地使用 Java 丰富的链接库。除此以外，使用 JPython 可以在 Java 中使用 Python 快速、简便地进行开发，充分利用 Python 的灵活性和快速性。

1.3.2 .NET 平台下的 Python: Python for .NET 和 IronPython

.NET 是 Microsoft XML Web services 平台，是微软公司极力推崇的技术。使用 XML Web services，不同的应用程序可以进行通信和数据共享。即使这些应用程序运行在不同的操作系统上，也不会影响它们之间的通信和数据共享。

随着 .NET 的流行，出现了 Python for .NET 和 IronPython 这两款 .NET 平台上的 Python。Python for .NET 可以使用 Python 与 .NET 进行交互。使用它可以在 .NET 中使用 Python 作为脚本语言，而且 Python 也可以通过它使用 .NET 中的服务。

IronPython 也是 Python 编程语言在 .NET 平台上的实现。IronPython 提供了和 Python 一样的交互式命令行。在交互式命令行下可以使用 Python 访问所有的 .NET 库。它除了对 Python 语言完全兼容以外，还支持所有的 .NET 类库，并且继承了 .NET Framework 的优点。通过它可以使用 Python 来扩展 .NET。

与 Python for .NET 相比，IronPython 的功能要更强大。IronPython 支持静态编译，通过静态编译，可以将 IronPython 程序编译成常规的 .NET 可执行文件。另外还可以将 IronPython 程序静态编译为 .NET 动态链接库，所编译的动态链接库可以被 C#、VB.NET 等调用。

不管是 Python for .NET 还是 IronPython 都还不完善。但是，借助于开源的神奇力量，相信 Python 在 .NET 上能够完善地使用只是时间上的问题。而且，微软的 .NET 在开源社区也很受欢迎，甚至在 Linux 上也可以运行 .NET 程序。

1.4 搭建开发环境

Python 可以运行在多种操作系统上，本书以 Windows 为平台来介绍 Python。Python 官方网站提供了 Windows 下的安装程序，除了安装 Python 以外，还应该选择用于编辑 Python 脚本的文本编辑器，提高编辑脚本的效率。因为在 Python 中使用缩进表示语句块，所以选择的文本编辑器要具有自动缩进的功能。

1.4.1 对操作系统的要求

Python 是跨平台的，可以运行在如下几种操作系统中：

- Windows;

- Linux/UNIX;
- Mac OS X;
- OS/2;
- Amiga。

另外, Python 还可以运行在一些掌上计算机和手机中。普通的用户一般不需要担心自己所使用的系统不能运行 Python, 因为绝大多数的流行操作系统都可以使用 Python。本书以 Windows XP 操作系统为平台, 主要使用 Python 2.5 进行讲解。

1.4.2 下载和安装 Python

Python 的安装程序以及源代码可以从其官方网站 <http://www.python.org> 获取。以 Windows XP、Python 2.5 为例, 在 Windows 下安装 Python 的过程如下。

(1) 从 Python 官方网站 <http://www.python.org> 下载 Python 在 Windows 下的安装程序 (python-2.5.1.msi)。

(2) 双击运行安装程序, 如图 1-2 所示。

(3) 如果系统中存在多个用户, 而其他用户并不需要使用 Python, 可以选择【Install just for me】; 否则可以按照默认的选项。

(4) 单击【Next】按钮, 如图 1-3 所示。此处可以按照默认的安装路径, 也可以根据需要进行选择 Python 的安装路径。

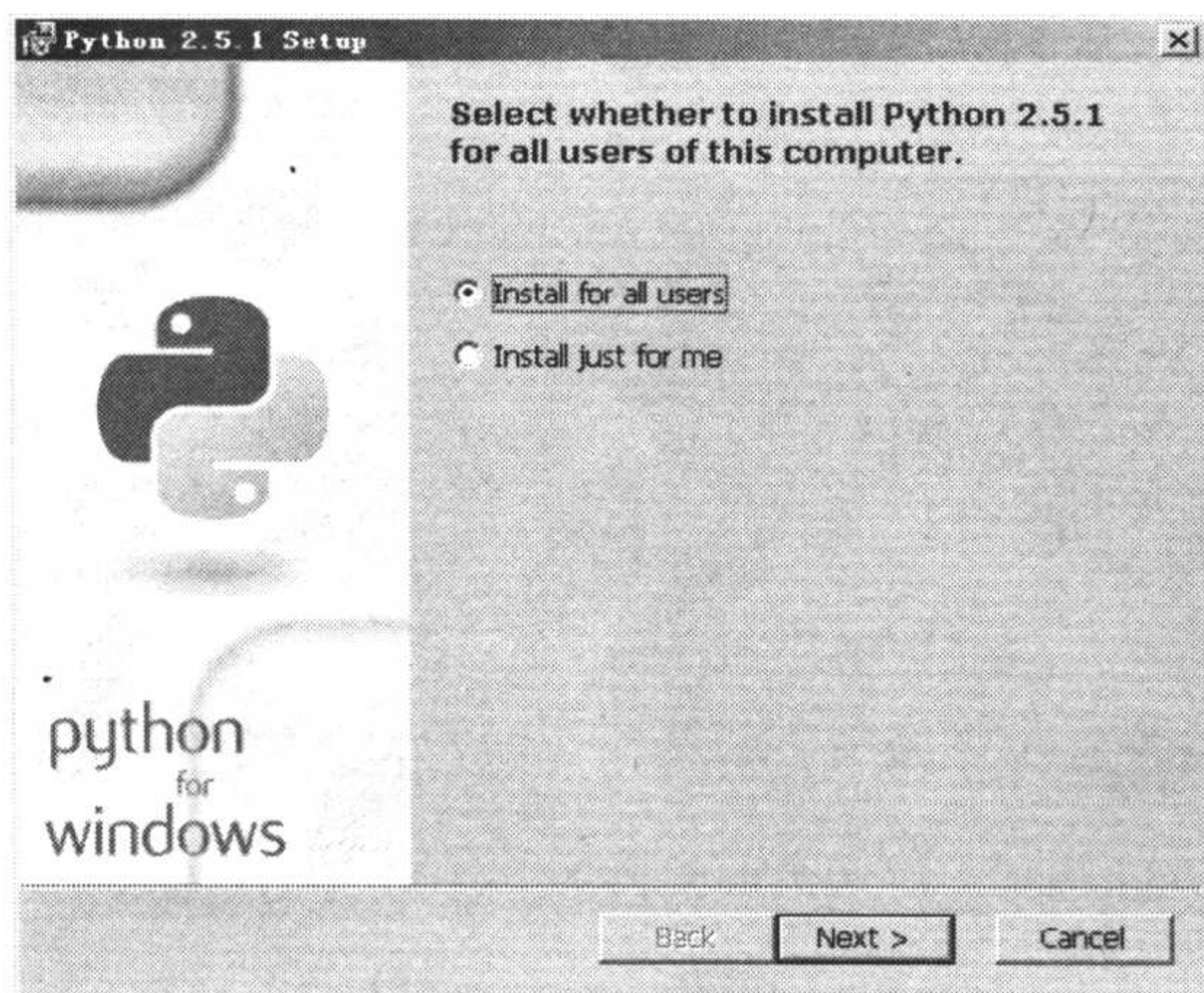


图 1-2 Python 安装程序



图 1-3 选择安装路径

(5) 单击【Next】按钮, 进入选择安装模块界面, 如图 1-4 所示。此处不需要做修改, 按照默认配置即可。

(6) 单击【Next】按钮，Python 安装程序将开始复制安装文件，如图 1-5 所示。

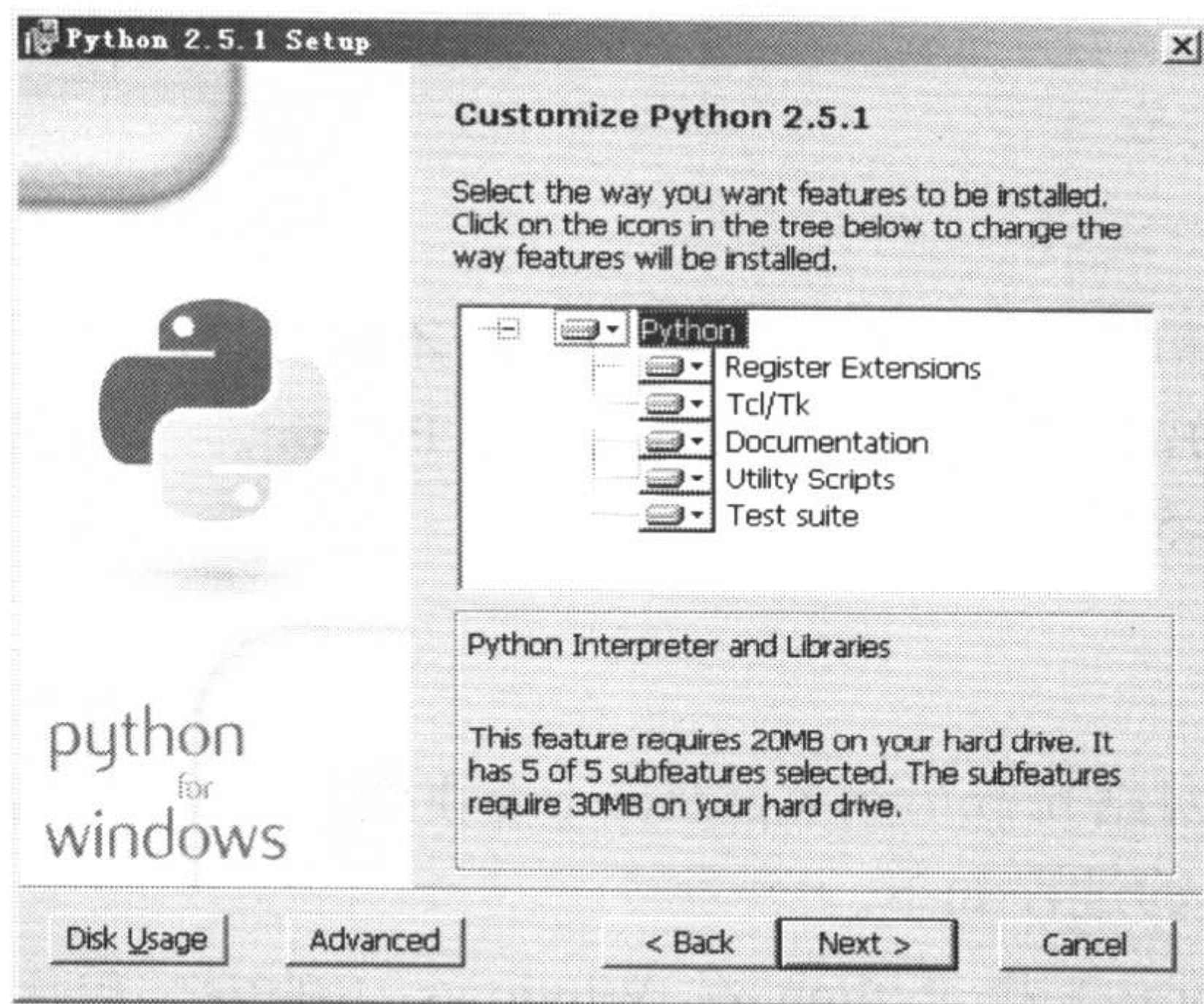


图 1-4 选择安装模块

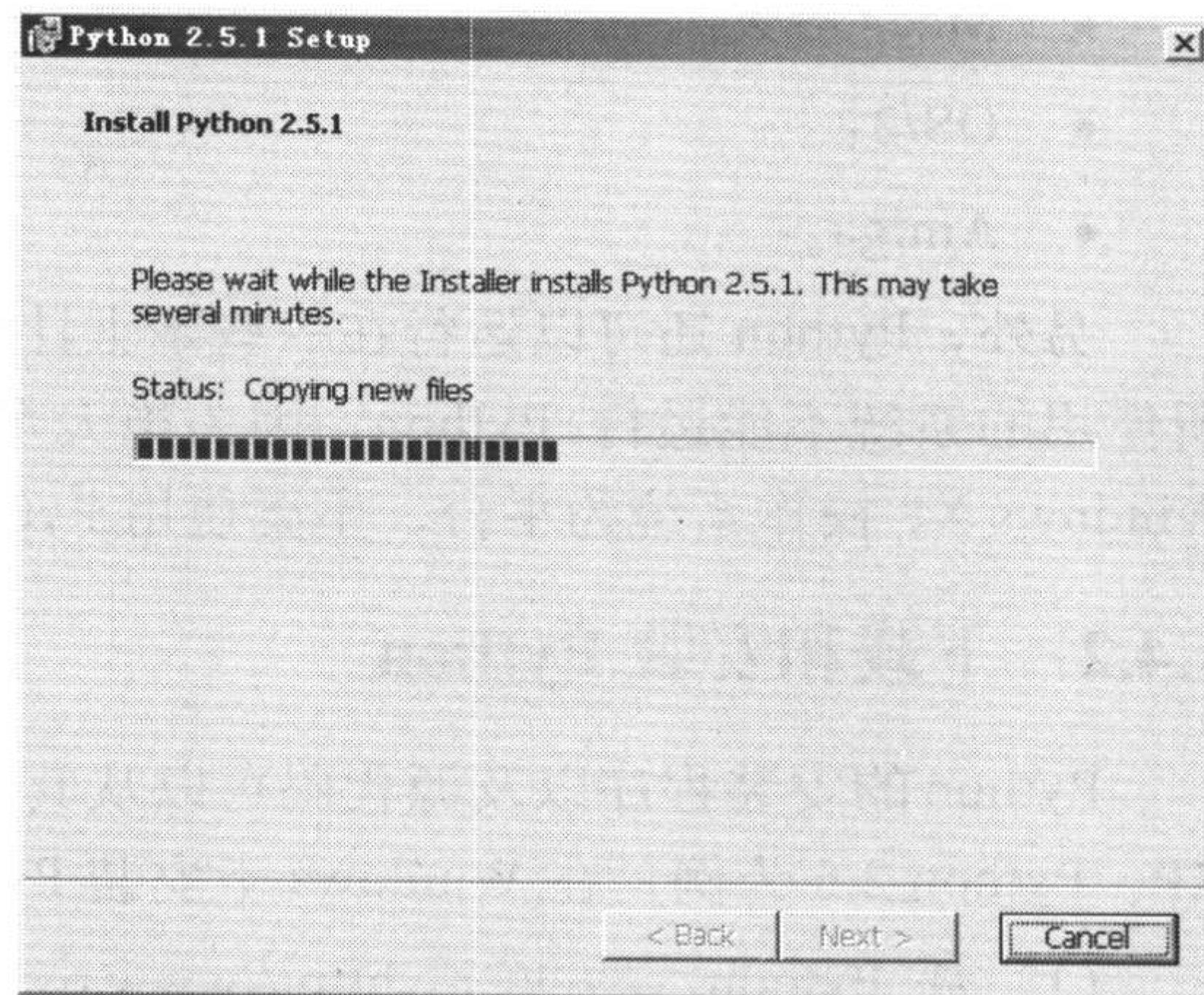


图 1-5 复制文件

(7) 当 Python 安装程序复制完文件以后，将进入如图 1-6 所示的界面。单击【Finish】按钮，将弹出对话框，要求重新启动系统。重新启动系统后即可完成 Python 安装。

(8) 安装完 Python 后，可以单击【开始】|【所有程序】|【Python2.5】|【Python (command line)】命令，将弹出如图 1-7 所示的 Python 的交互式命令行界面。也可以单击【开始】|【运行】命令，在弹出的对话框中输入“python”即可。

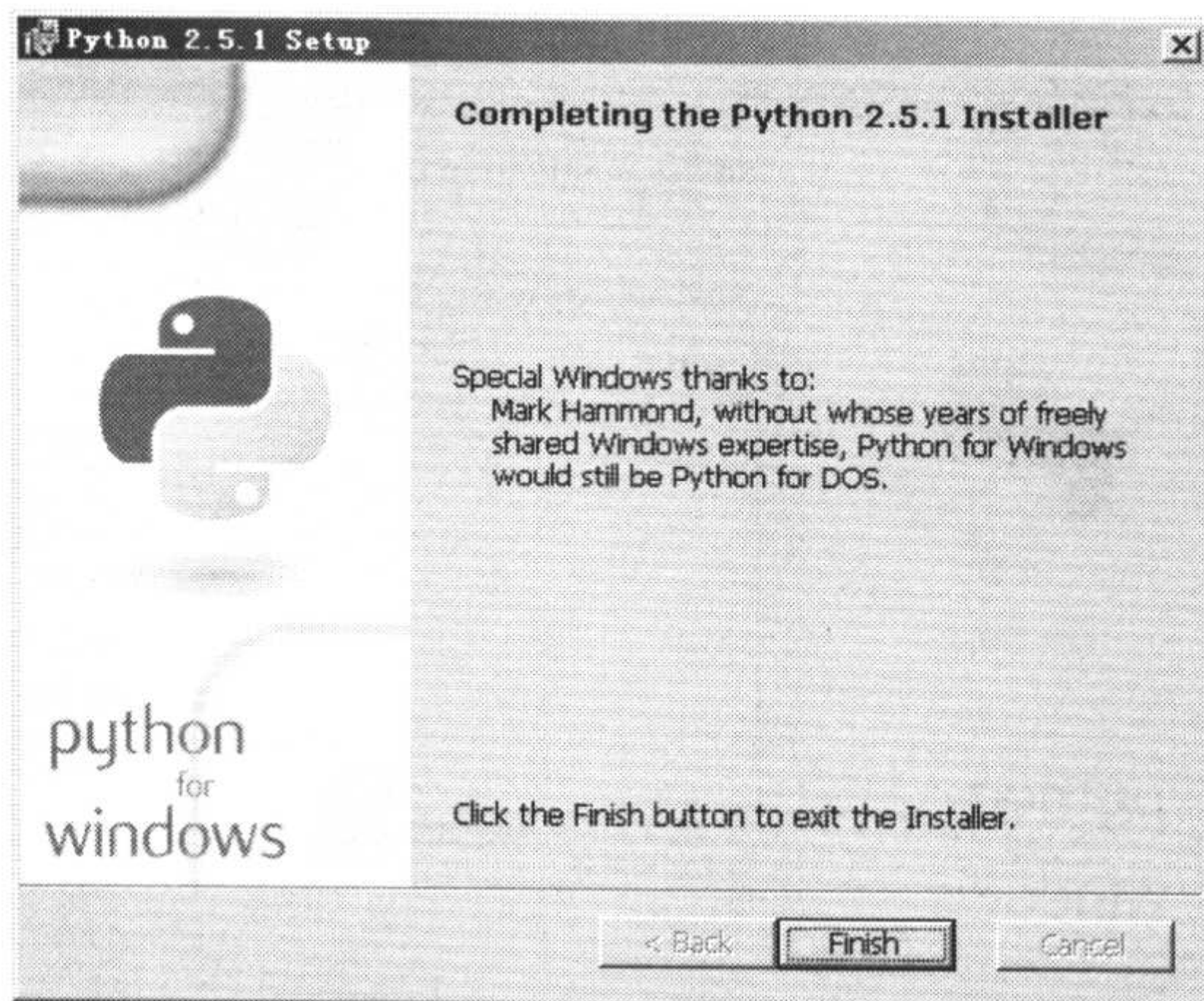


图 1-6 安装完成

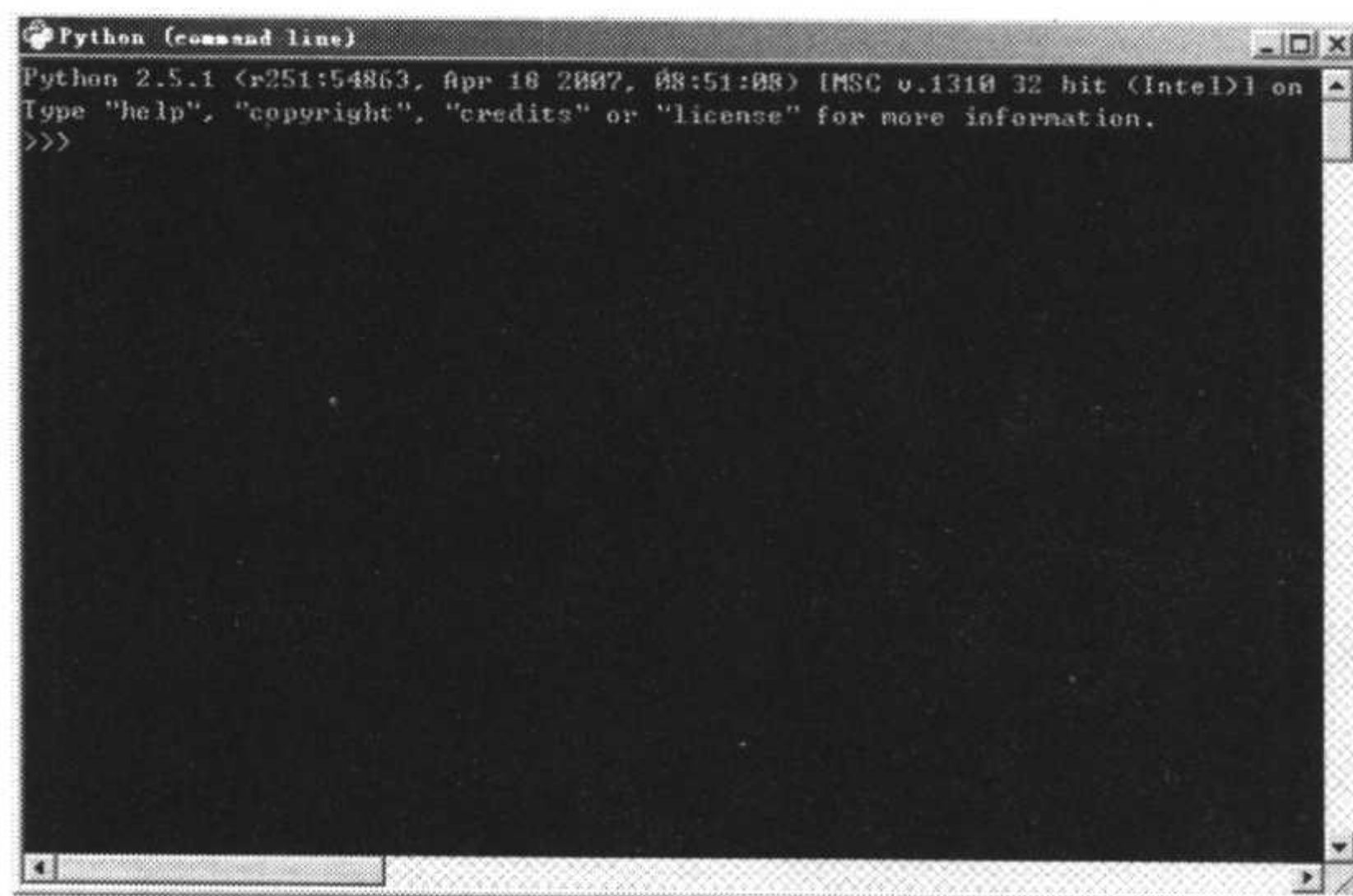


图 1-7 Python 交互式命令行

如果没能打开 Python 的交互式命令行，可以将 Python 的安装路径添加到“path”系统变量中。右键单击【我的电脑】图标，选择菜单【属性】命令，在弹出的对话框中选择【高级】标签，如图 1-8 所示。单击【环境变量】按钮，将弹出如图 1-9 所示的对话框。选中【用户变

第1章 Python 概述

量】中的“path”选项，单击【编辑】按钮，将弹出如图 1-10 所示的对话框。在【变量值】文本框中的末尾添加“;D:\Python25”，单击【确定】按钮。然后重新启动系统即可。

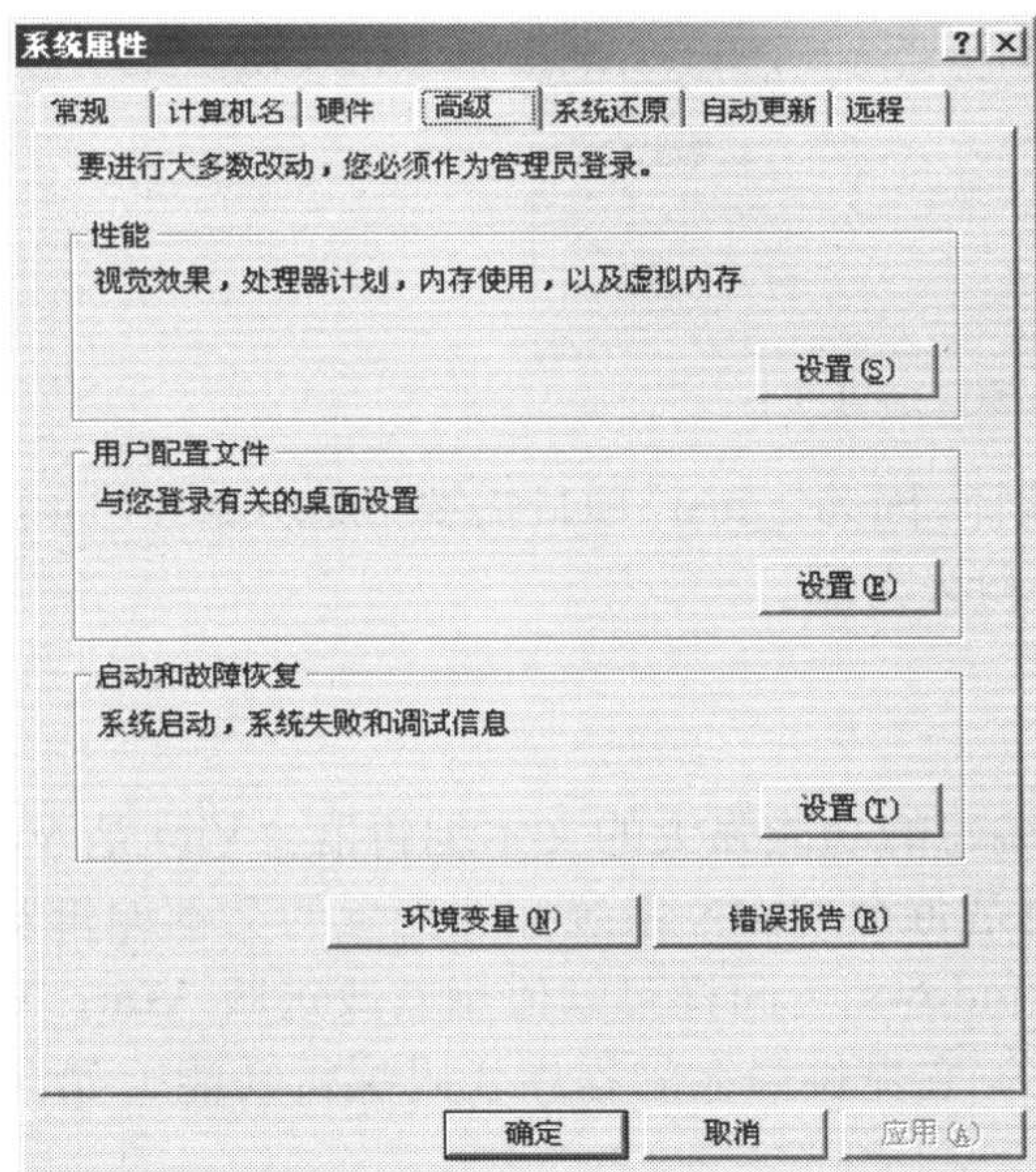


图 1-8 【高级】标签

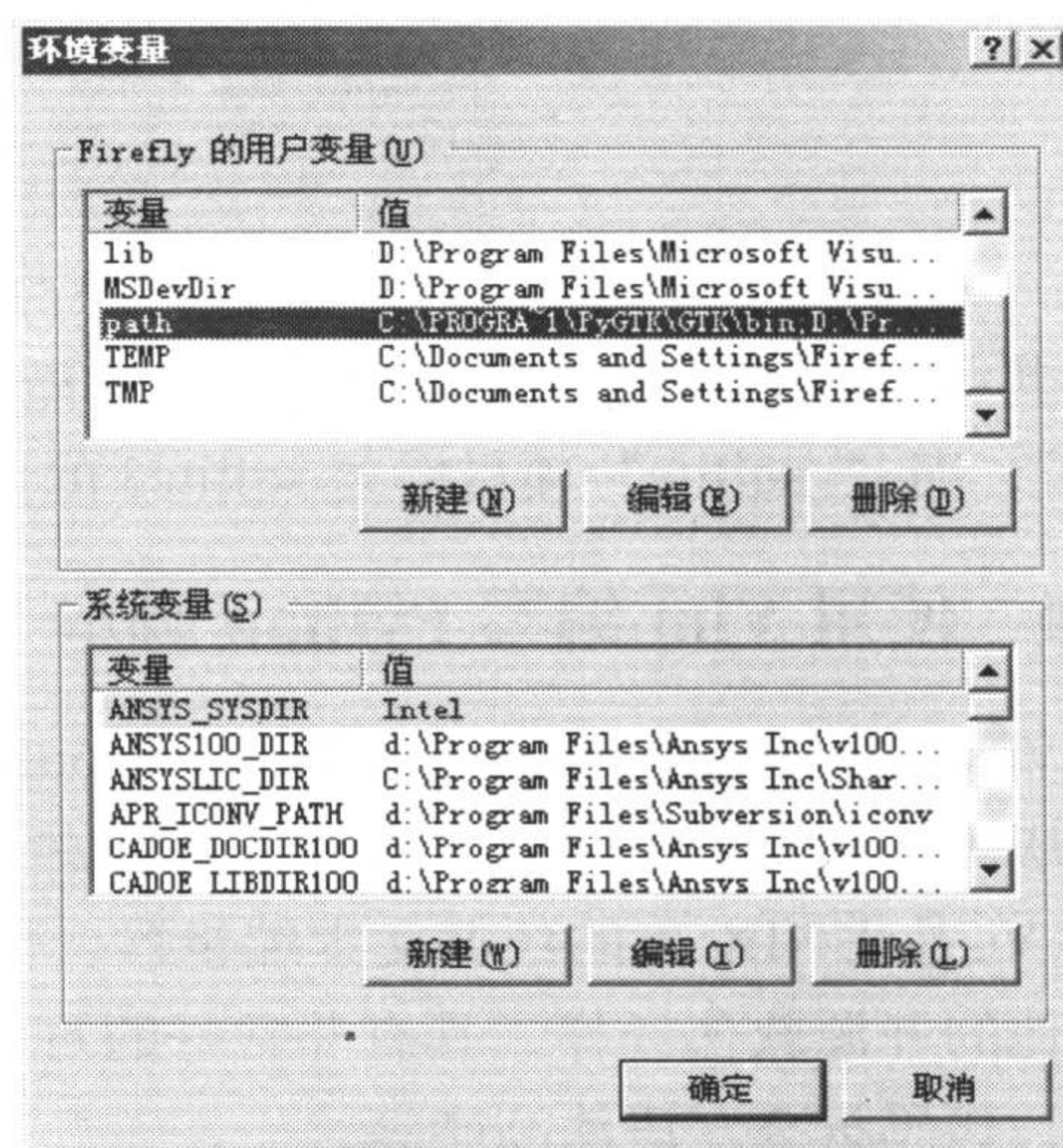


图 1-9 环境变量

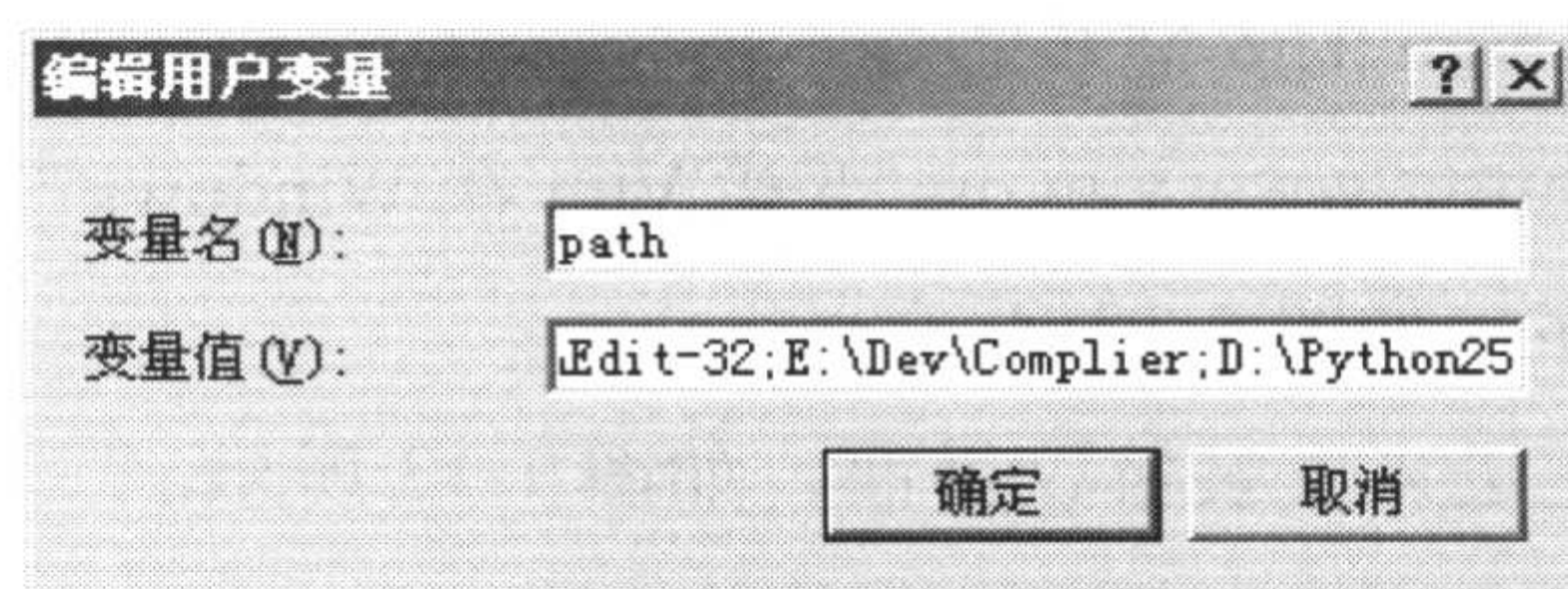


图 1-10 编辑用户变量

1.4.3 自己编译 Python

由于 Python 提供了源代码，因此可以自己编译 Python。在 Windows 平台下可以使用从 Visual C++ 6.0 到最新的 Visual Studio 2005 的任何软件编译 Python。以 Visual Studio 2005 为例，在 Windows 下编译 Python 的过程如下。

- (1) 从 Python 官方网站下载 Python 的源代码压缩包 Python-2.5.1.tar.bz2。
- (2) 将 Python 源代码解压至某一目录下，使用 Visual Studio 2005 打开其中的“PCbuild8”目录下的“pcbuild.sln”文件。
- (3) 单击菜单【生成】|【批生成】命令，将弹出批生成对话框。仅把 Win32 平台下的项目选中，如图 1-11 所示。单击【生成】按钮，即可编译 Python。

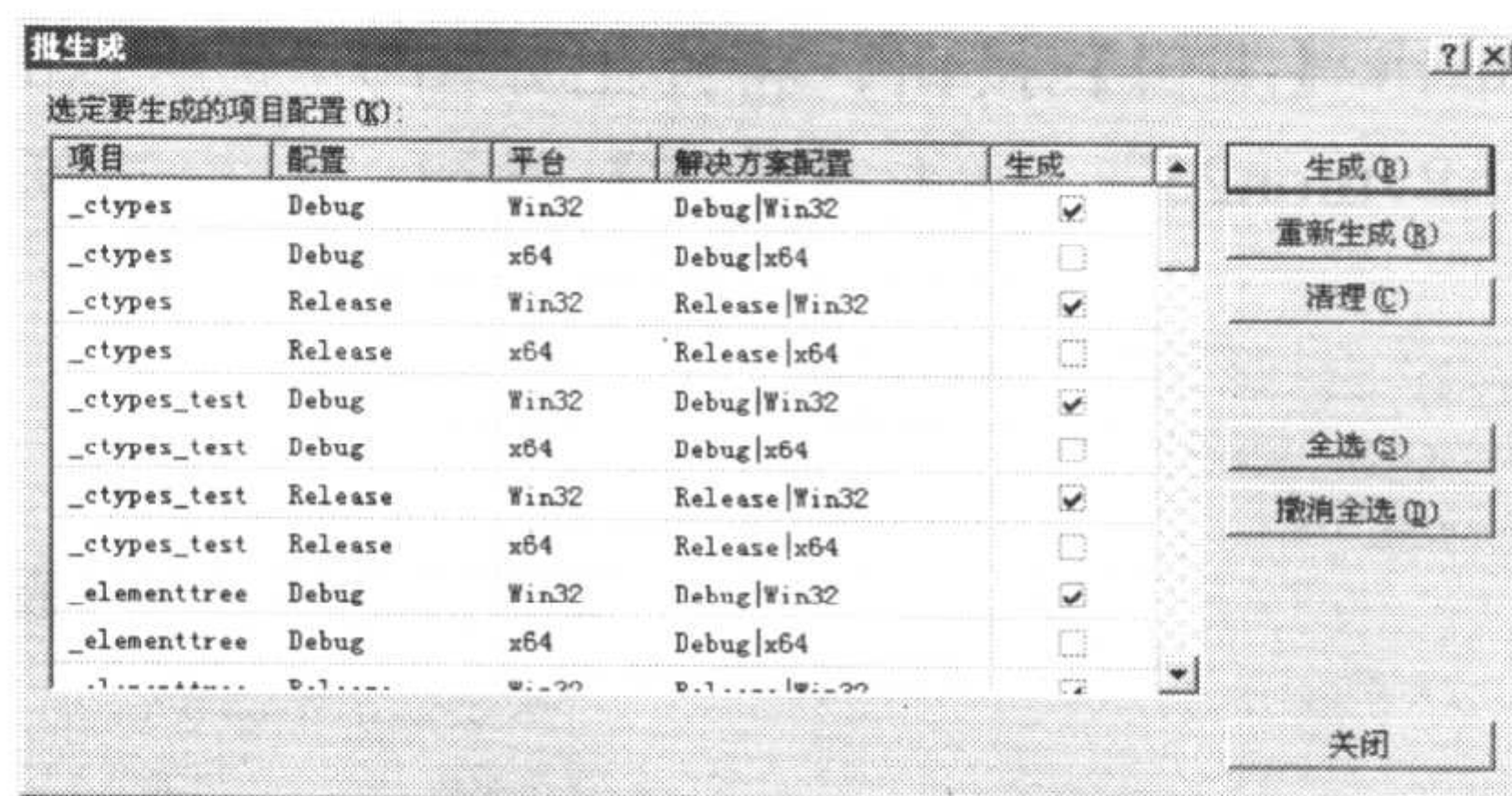


图 1-11 选择编译项目

其中“_msi”项目和“_sqlite”项目不能成功编译,因为“_msi”项目需要“Microsoft Platform SDK”,而“_sqlite”项目需要 sqlite3 的库文件和头文件。

1.4.4 使用 Vim 编写 Python 脚本

Vim (Vi Improved) 是由 Bram Moolenaar 编写的功能强大的文本编辑器。Vim 是 Vi 的改进版本,Vi 是 UNIX 下经典的文本编辑器,也是每个 UNIX 系统标准配置的文本编辑器。虽然 Vi 是 UNIX 下的经典文本编辑器,但在 Windows 下同样可以使用其改进版 Vim。Vim 和其他的开源软件一样,具有非常好的可移植性,它提供强大的程序代码编辑功能,如自动缩进、代码折叠、语法高亮等。

1. 安装 Vim

以 Windows XP 为例, Vim 的安装步骤如下所示。

- (1) 从 Vim 官方网站 <http://www.vim.org> 下载 Vim 在 Windows 下的安装程序 gvim71.exe。
- (2) 双击运行安装程序,如图 1-12 所示。
- (3) 单击【是】按钮,进入安装协议界面,如图 1-13 所示。

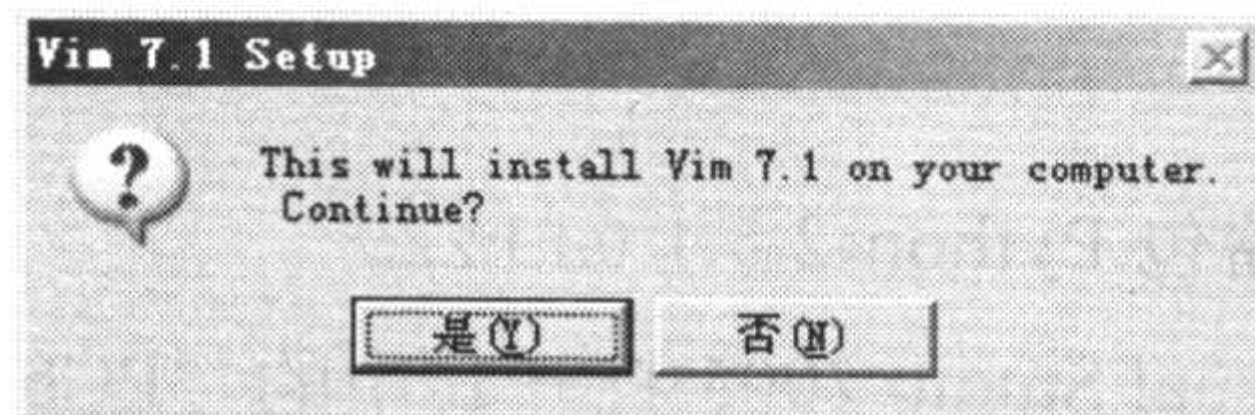


图 1-12 Vim 安装程序

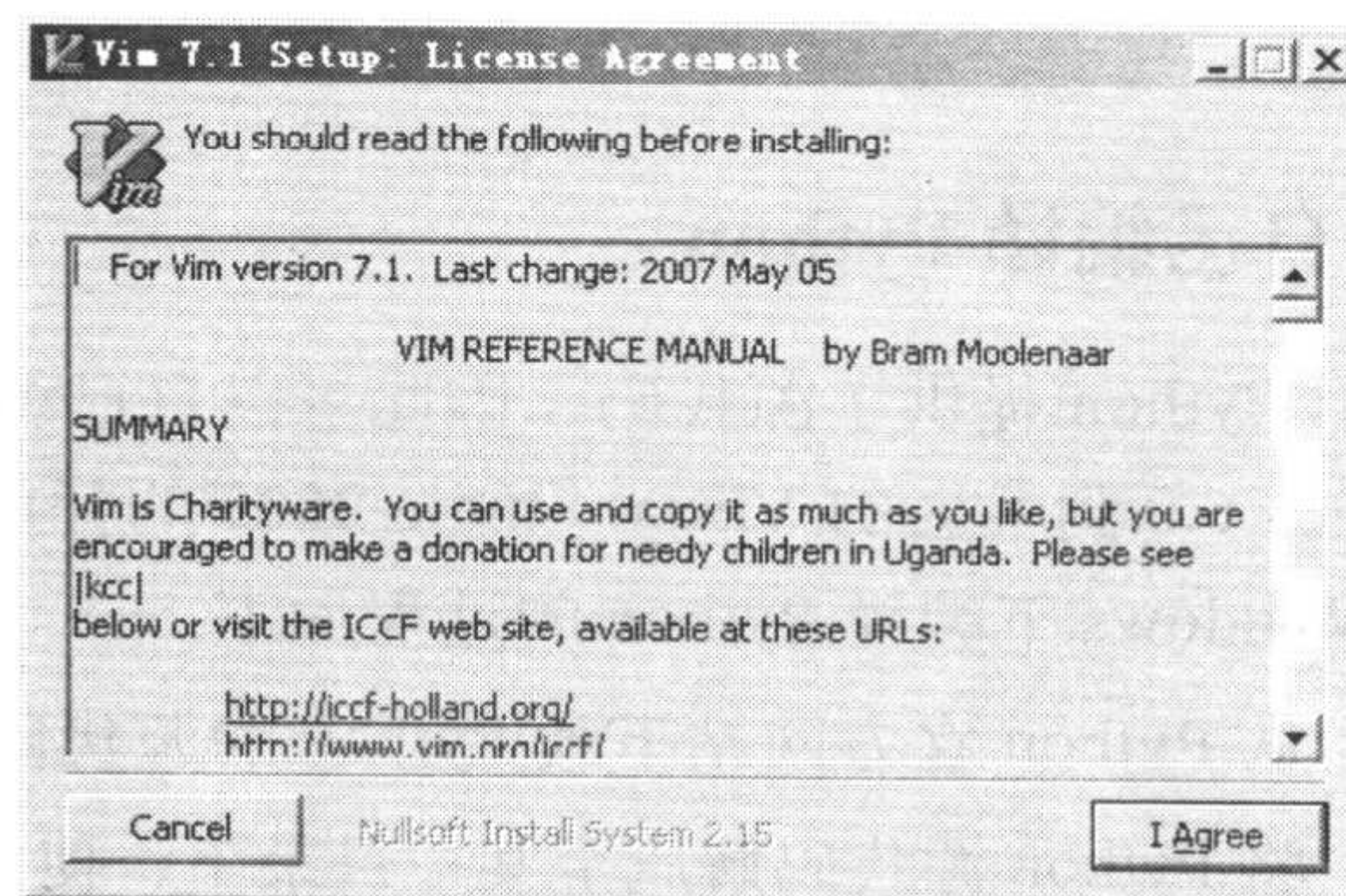


图 1-13 安装协议

(4) 单击【I Agree】按钮,进入安装模块选择界面,如图 1-14 所示。可以选中【Create .bat files for command line use】复选框,这样可以在 Windows 的命令行下使用 Vim。

(5) 单击【Next】按钮，将开始安装 Vim。在安装过程中将弹出如图 1-15 所示的命令行窗口，需要按回车键。当 Vim 安装程序复制完文件后如图 1-16 所示。

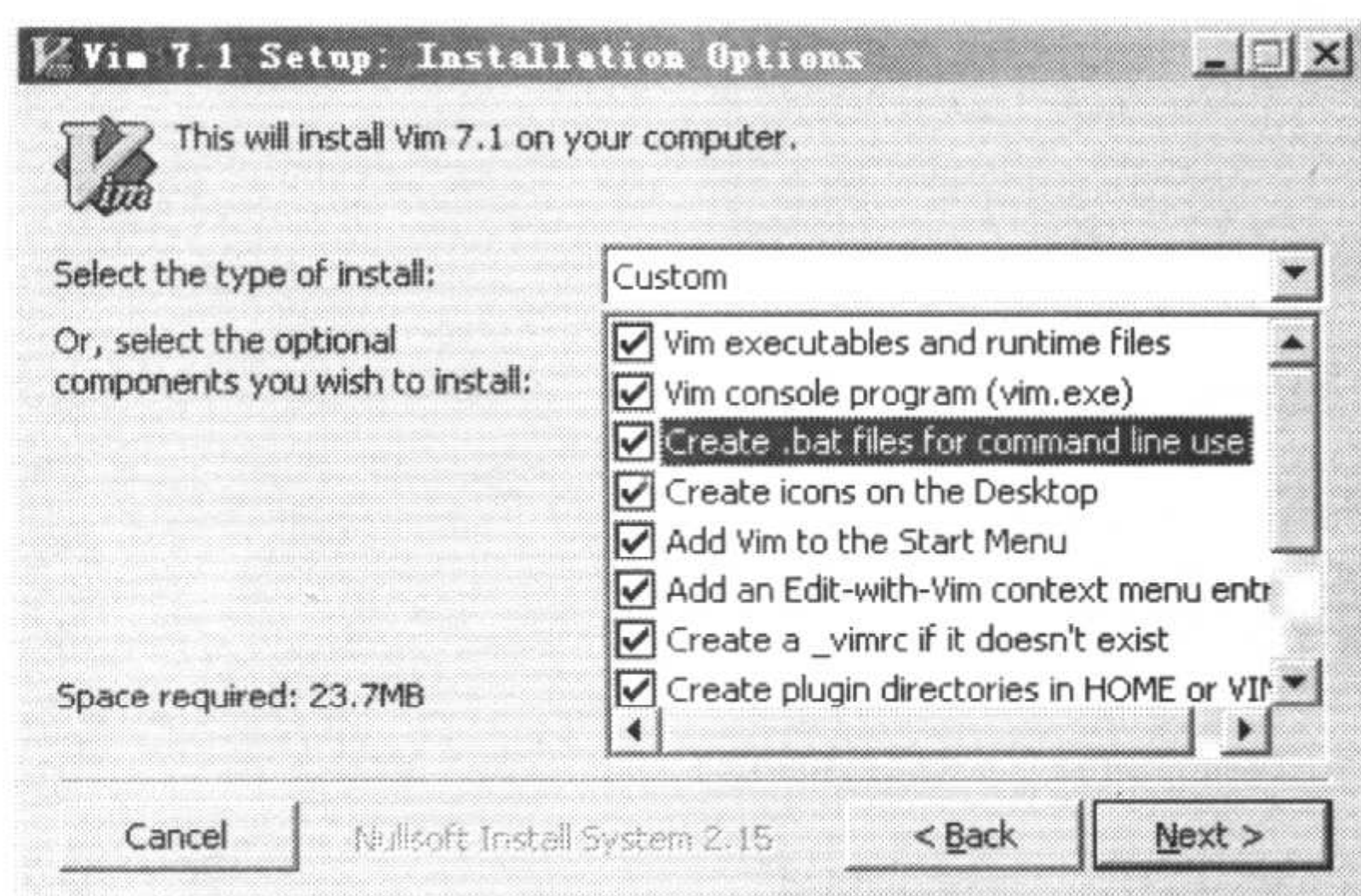


图 1-14 选择安装模块



图 1-15 命令行窗口

(6) 单击【Close】按钮，将弹出如图 1-17 所示的对话框，可以选择【否】按钮。当完成 Vim 安装后，将在桌面创建如下 3 个快捷方式。

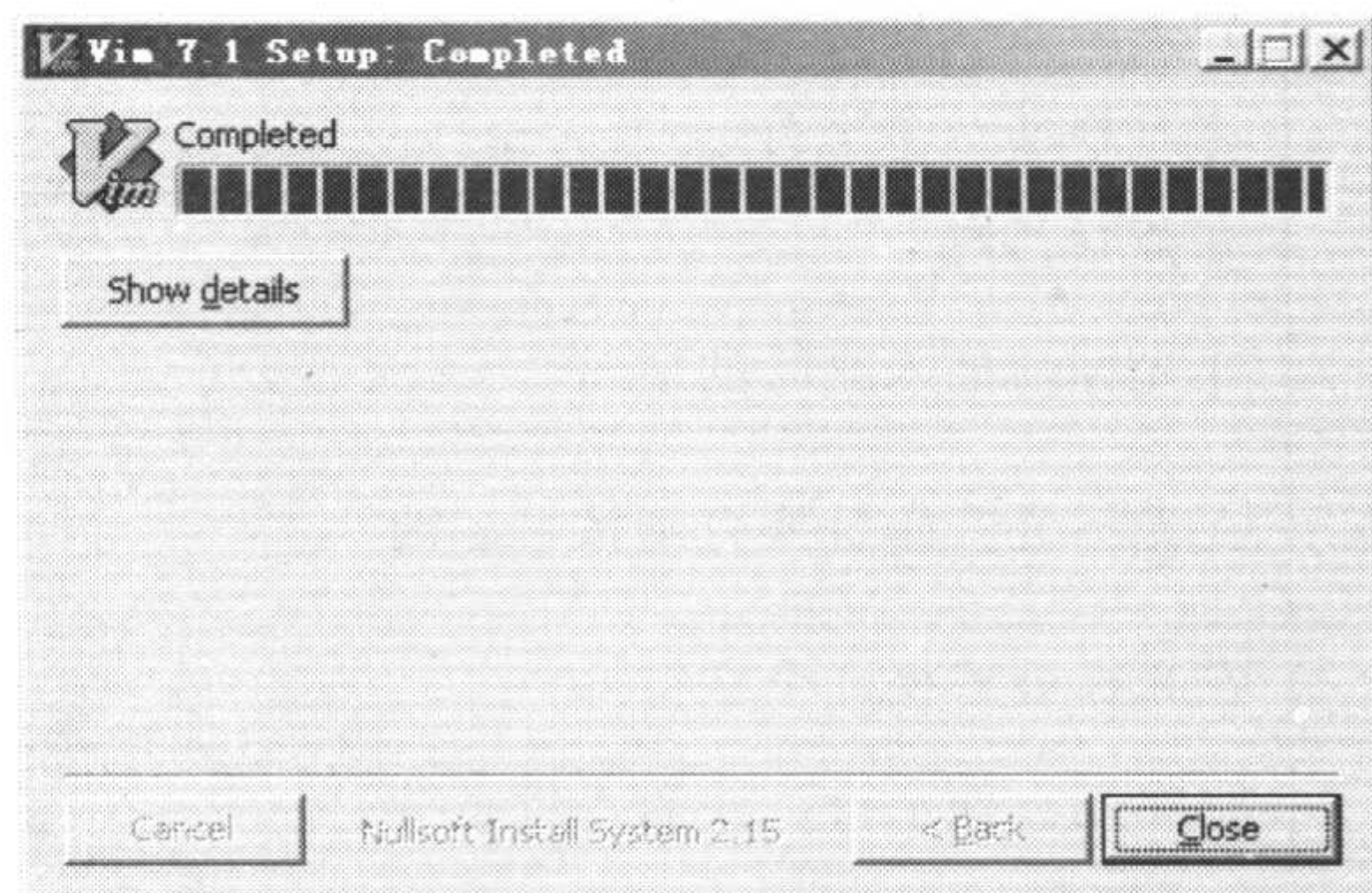


图 1-16 安装完成

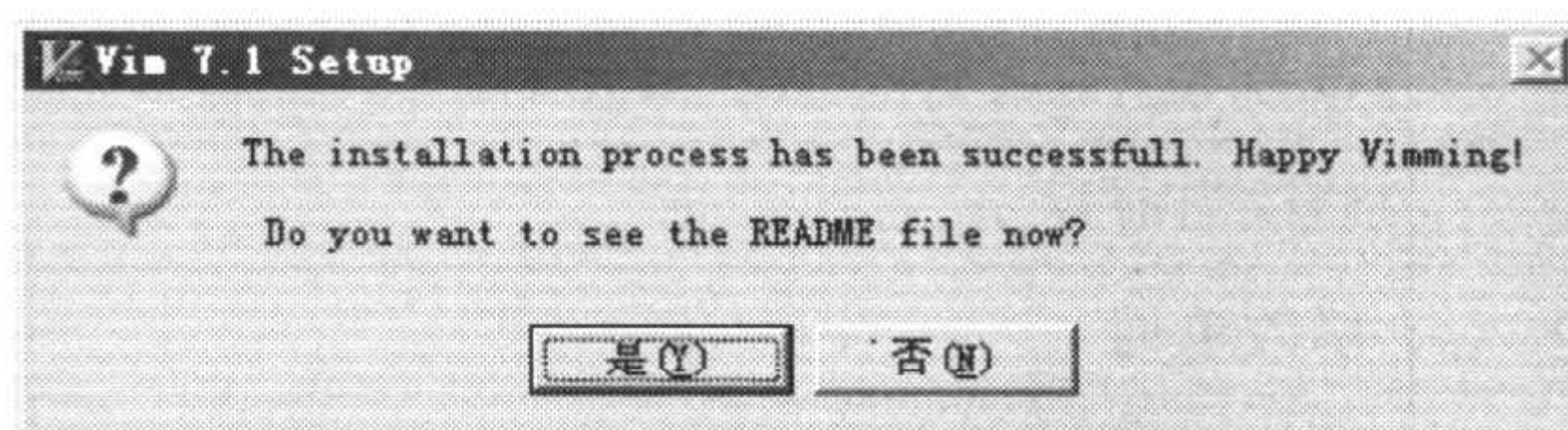


图 1-17 是否阅读 README 文件对话框

- gVim 7.1: 标准的 Vim。
- gVim Easy 7.1: 无模式的 Vim。
- gVim Read only 7.1: 制度模式的 Vim。

安装好 Vim 后，还需要打开 Vim 安装目录下的“_vimrc”文件对 Vim 进行简单地设置。在“_vimrc”文件中添加“set nobackup”可以不生成备份文件，添加“set nu”可以显示行号，添加“colo 配色方案名”可以修改默认配色方案。运行 Vim 后如图 1-18 所示。

2. Vim 命令简介

Vim 是有模式编辑器，它有两种模式：命令模式和编辑模式。在命令模式下输入的字符被解释为命令，在编辑模式下则像使用其他文本编辑器一样，如 Windows 的记事本。在编辑

模式下按 Esc 键可以回到命令模式。Vim 常用的命令如表 1-1 所示。

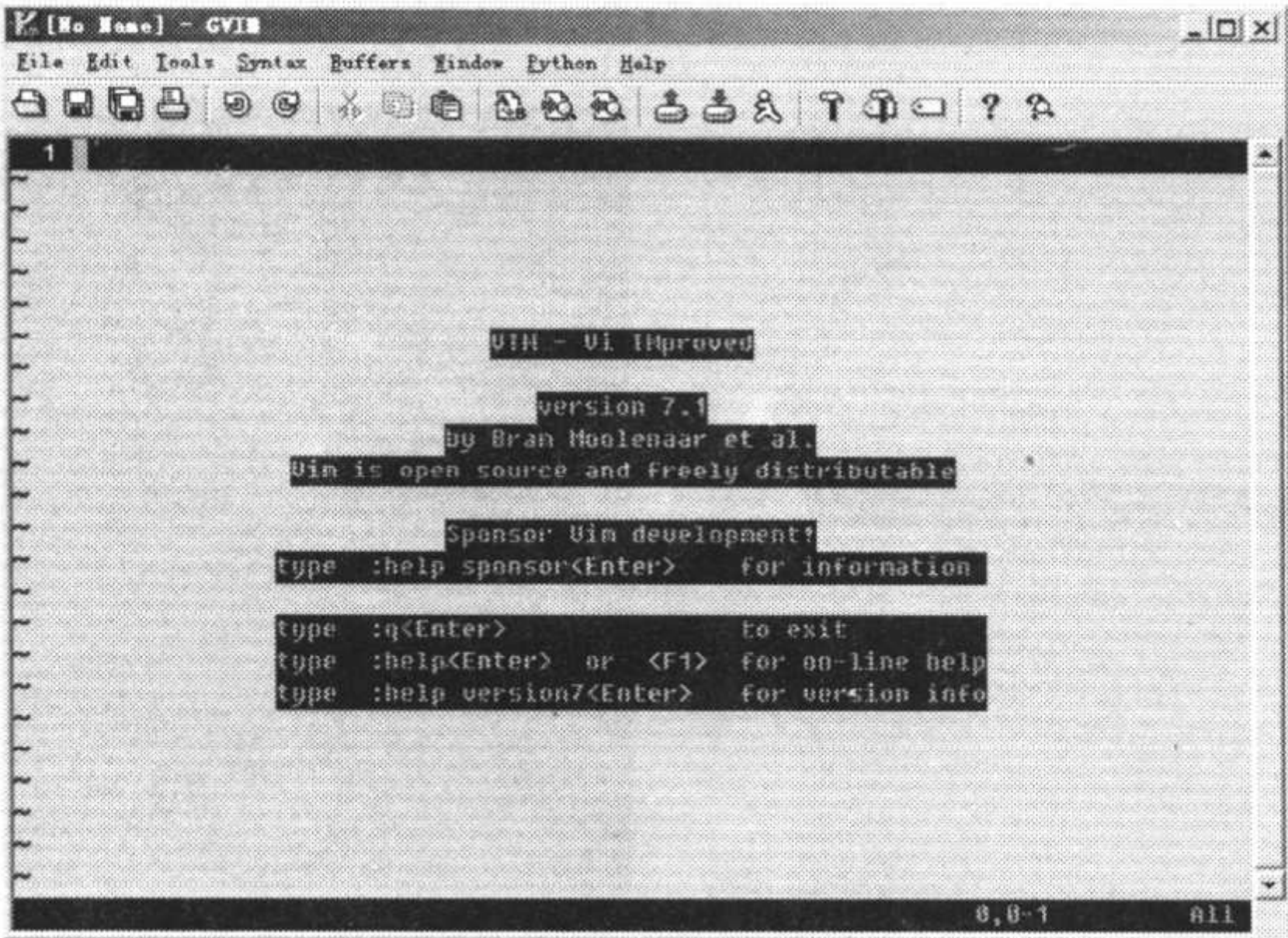


图 1-18 Vim 编辑器

表 1-1 Vim 的常用命令

命令	说 明	命令	说 明
~	转换大小写	k	向上移动一行
A	在行尾追加字符	l	向右移动一个字符
a	在光标之后追加字符	o	新建空白行
b	向后移动一个单词，至单词首字母	p	在光标前插入删除的字符
B	向后移动一个单词，至单词首字母，忽略标点符号	R	在光标处开始替换直到按 Esc 键
cc	修改当前行	r	在光标处替换一个字符
cw	修改单词	s	删除光标处字符，并插入新字符
D	从光标处删除至行尾	S	删除当前行，并插入新字符
dd	删除当前行	u	撤销
dw	删除单词	V	进入块模式（在 Windows 下需要按 Ctrl+Q 组合键）
e	向前移动至单词尾	v	进入行模式
E	向前移动至单词尾，忽略标点符号	w	向前移动一个单词，至词首
h	向左移动一个字符	W	向前移动一个单词，至词首，忽略标点符号
i	在光标处插入字符	x	删除光标处字符
I	在行首不为空白处插入字符	yw	复制一个单词
j	向下移动一行	yy	复制一行
J	合并两行		

第1章 Python 概述

上述所列举的命令都是在命令模式下输入的命令。如果在这些命令前加上数字，则表示重复执行多少次命令。例如，在命令模式下输入“3dd”，将删除从当前行开始共3行文本。

另外在编辑模式下有几个对编程非常有用的命令，如 Ctrl+P（先按下 Ctrl 键再按下 P 键）组合键或者 Ctrl+N 组合键可以补全当前单词（前提是该单词已经在当前 Vim 编辑的文件中出现过），如图 1-19 所示。如果所安装的 Vim 支持 Python，在编辑 Python 脚本时还可以通过按 Ctrl+X 组合键，然后按 Ctrl+O 组合键自动补全 Python 模块中的函数或者属性，如图 1-20 所示。如果所安装的 Vim 不支持 Python，即不能使用 Ctrl+X、Ctrl+O 组合键则需要自己编译 Vim。Vim 还支持正则表达式，使用正则表达式可以完成非常复杂的查找替换工作。

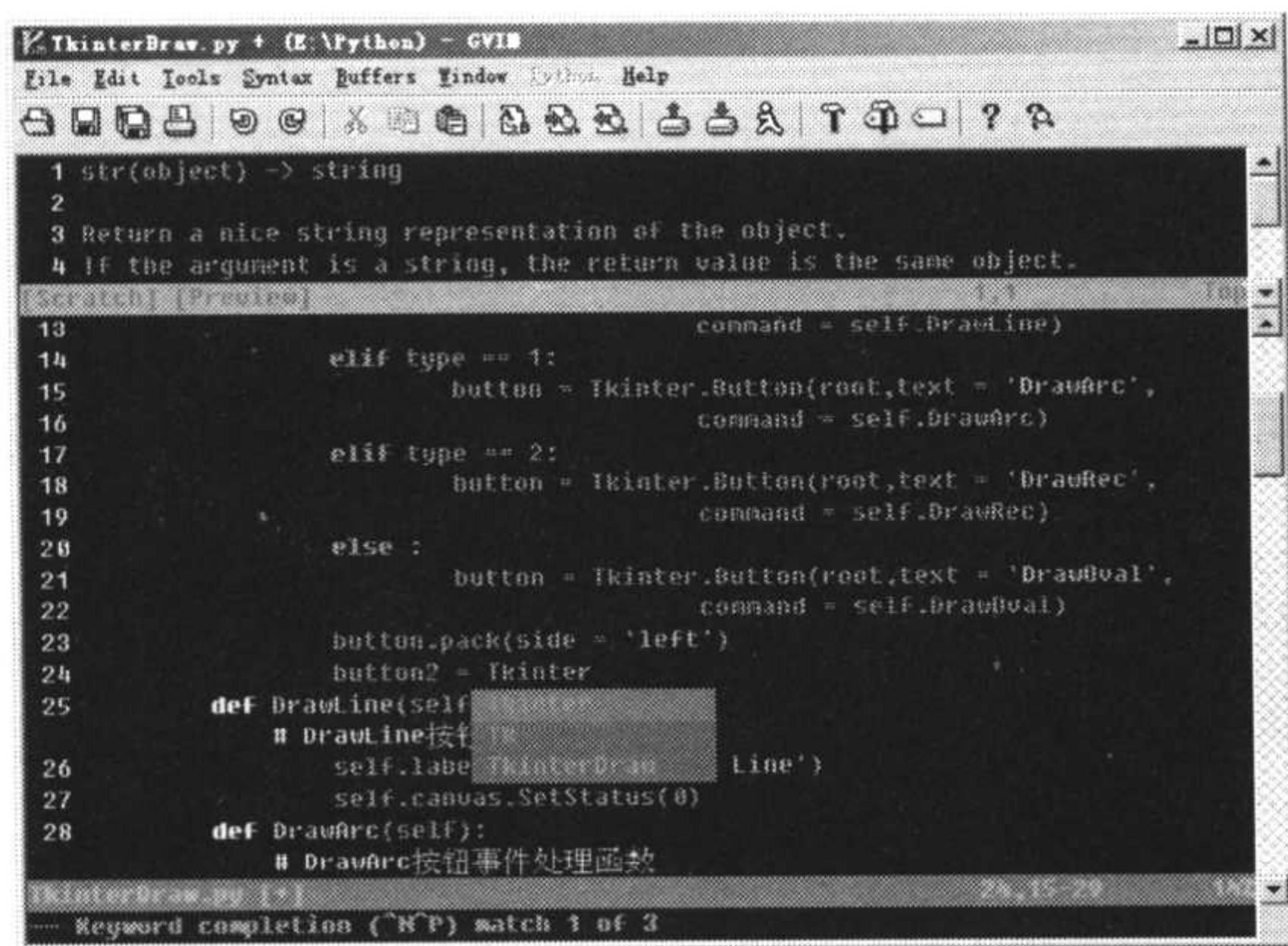


图 1-19 自动补全

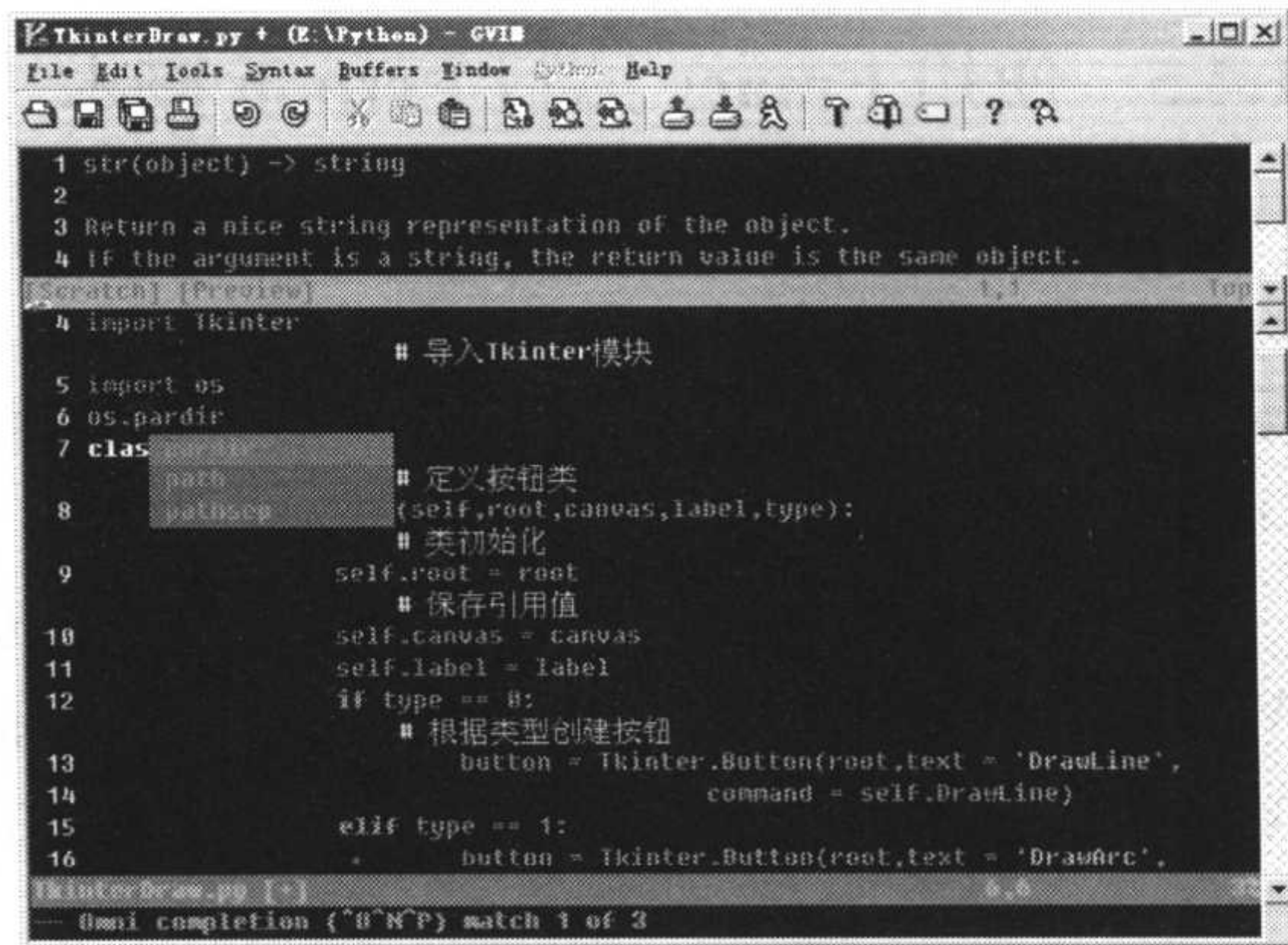


图 1-20 自动补全模块中的函数

如果需要自己编译 Vim 实现模块函数自动补全也很容易。以 Visual Studio 2005 为例，自己编译 Vim 的过程如下所示。

(1) 从 Vim 官方网站下载 vim-7.1.tar.bz2 和 vim-7.1-extra.tar.gz 压缩包，将解压后的两个目录中的“vim71”目录合并。

(2) 单击【开始】|【所有程序】|【Microsoft Visual Studio 2005】|【Visual Studio Tools】|【Visual Studio 命令提示】命令，将弹出如图 1-21 所示的命令行窗口。

(3) 从命令行中进入“vim71”目录下的“src”目录。

(4) 在命令行中输入如下所示的命令编译 Vim。

```
nmake -f Make_mvc.mak GUI=yes MBYTE=yes PYTHON=D:\Python25 PYTHON_VER=25
```

(5) 编译完成后，可以将编译好的 gvim.exe 文件直接复制到已安装的 Vim 目录中，将原有的 gvim.exe 文件覆盖。

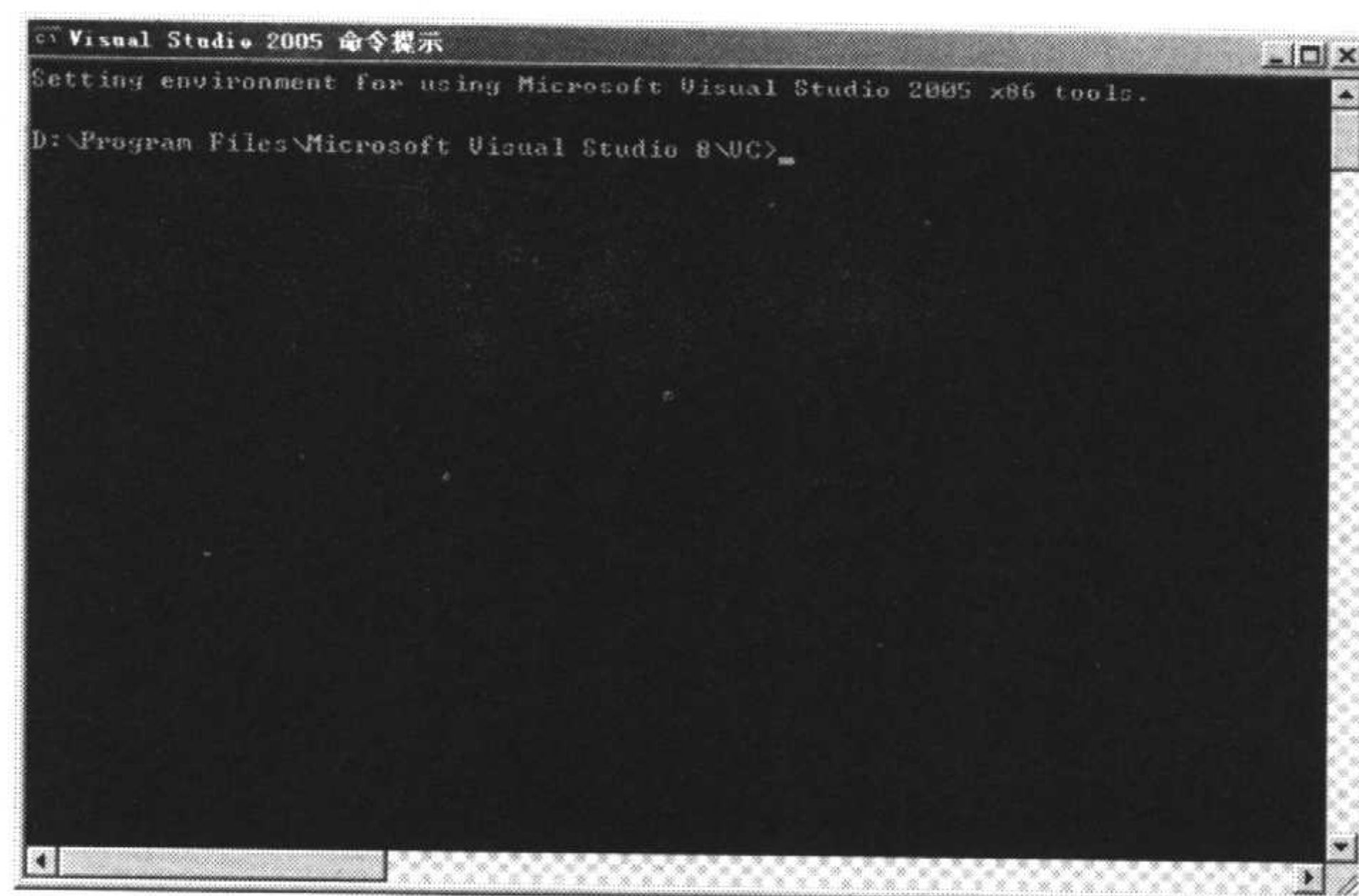


图 1-21 Visual Studio 命令提示

1.4.5 使用 Emacs 编写 Python 脚本

Emacs 号称是世界上最强大的文本编辑器，Emacs 被设计得“无所不能”。世界上讨论最多的文本编辑器是 Vim 和 Emacs，而且关于是否选用 Vim 还是 Emacs 的争论从来就没有停止过。与 Vim 不同，Emacs 不是有模式编辑器。使用 Emacs 就像使用 Windows 的记事本一样，但 Emacs 要强大得多。

在 Windows 下安装 Emacs 可以从 <http://ntemacs.sourceforge.net> 网站下载编译好的 Windows 安装程序。该安装程序是一个自解压的压缩文件，只需选择解压目录即可。解压完成后，运行 Emacs 所在目录下“bin”目录中的“runemacs.exe”文件，如图 1-22 所示。

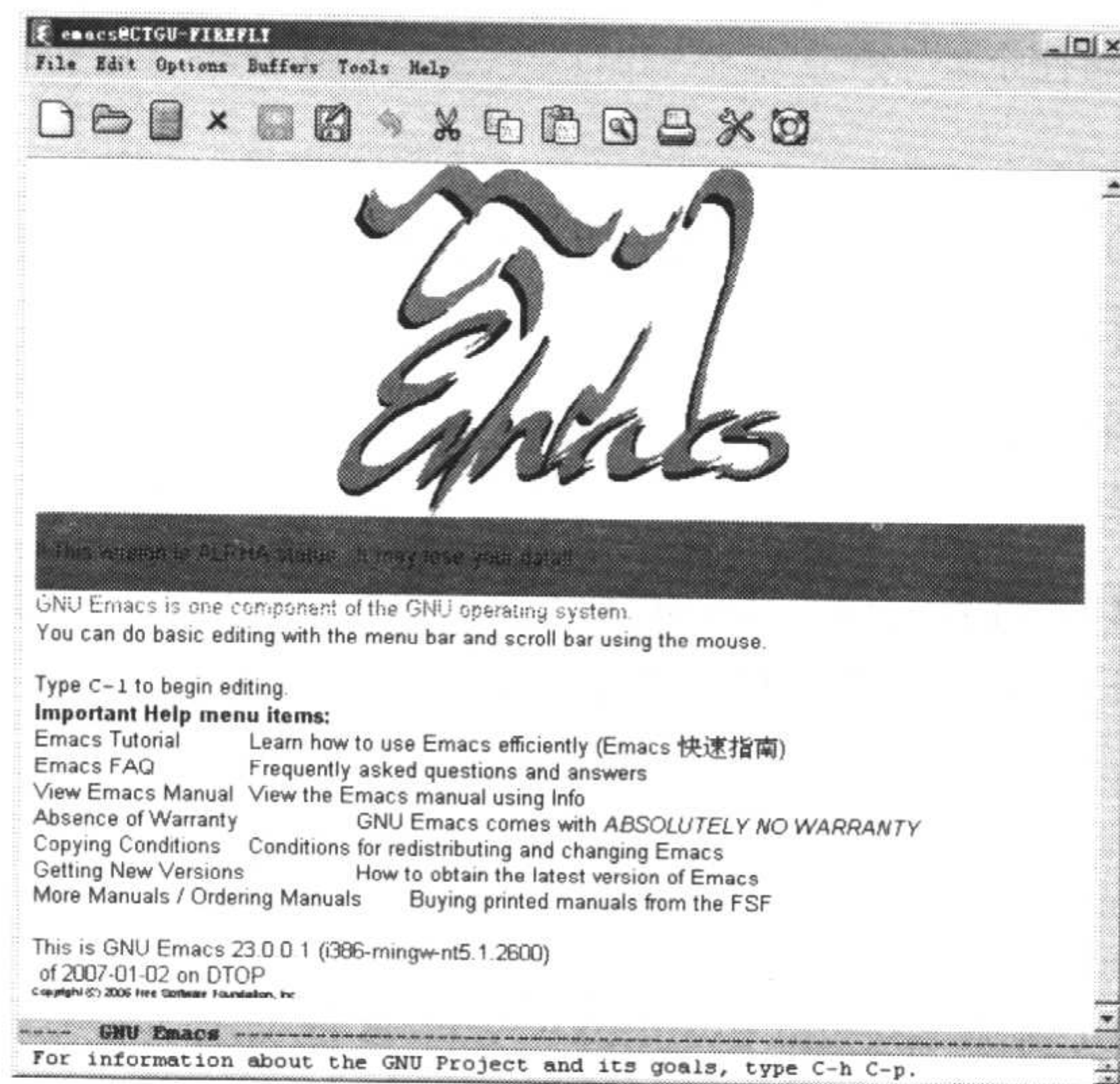


图 1-22 Emacs 文本编辑器

Emacs 中常用的命令如表 1-2 所示。

表 1-2

Emacs 中常用的命令

命 令	说 明	命 令	说 明
C-v	向后翻一页	M-p	选择前一个查找字符串
M-v	向前翻一页	M-n	选择下一个查找字符串
C-l	将当前行居中	C-d	向前删除字符
C-f	向前移动一个字符	M-d	向前删除到字首
M-f	向前移动一个单词	M-DEL	向后删除到字尾
C-b	向后移动一个字符	M-0 C-k	向前删除到行首
M-b	向后移动一个单词	C-k	向后删除到行尾
C-n	向下移动一行	C-x DEL	向前删除到句首
C-p	向上移动一行	M-k	向后删除到句尾
C-a	移至当前行的第一个字符	M-- C-M-k	向前删除到表达式首部
M-a	移至当前所在句子的第一个字符	C-M-k	向后删除到表达式尾部
C-e	移至当前行的最后一个字符	C-x r r	复制一个矩形到寄存器
M-p	移至当前所在句子的最后一个字符	C-x r k	删除矩形
M-<	移动到当前窗口的第一个字符	C-x r y	插入刚刚删除的矩形
M->	移动到当前窗口的最后一个字符	C-x r o	打开一个矩形，将文本移动至右边
C-x C-c	永久离开 Emacs	C-x r c	清空矩形
C-x C-f	读取文件到 Emacs	C-x r t	为矩形中每一行加上一个字符串前缀
C-x r	以只读方式打开一个文件	C-x r i r	从 r 缓冲区内插入一个矩形
C-x C-q	清除一个窗口的只读属性	C-x l	删除所有其他窗口
C-x C-s	保存文件到磁盘	C-x 2	上下分割当前窗口
C-x s	保存所有文件	C-x 3	左右分割当前窗口
C-x i	插入其他文件的内容到当前缓冲	C-x 0	删除当前窗口
C-x C-v	用将要读取的文件替换当前文件	C-M-v	滚动其他窗口
C-x C-w	将当前缓冲写入指定的文件	C-x o	切换光标到另一个窗口
C-s	向前查找	C-x ^	增加窗口高度
C-r	向后查找	C-x {	减小窗口宽度
C-M-s	规则表达式查找	C-x }	增加窗口宽度
C-M-r	反向规则表达式查找		

需要说明的是命令中的“C”代表按下 Ctrl 键，“M”代表按下 Alt 键。例如命令“C-v”是指按着 Ctrl 键不放然后再按“v”键。而命令“C-x C-f”则指按住 Ctrl 键不放，先按下“x”键，然后松开“x”键再按“f”键。而“M-<”命令则指先按住 Alt 键不放，接着再按住 Shift 键不放，最后再按“<”，这是因为按住 Shift 键不放然后按“<”才是“<”。

1.4.6 使用 PythonWin 编写 Python 脚本

前面所提到的 Vim 和 Emacs 两种文本编辑器各有特色，但上手较难。尤其是 Emacs 在操作的时候需要使用大量的快捷键，初学者很难记住。但一旦习惯，并且深入地学习 Vim 和 Emacs 后，会发现它们确实非常强大。在 Windows 下还有一款比较好用的 Python 脚本编辑器——PythonWin 所附带的编辑器。PythonWin 是 Python 在 Windows 下的扩展包，使用 PythonWin 可以让 Python 使用 Windows 系统的特性。

1. PythonWin 安装

在 Windows 下使用 Python 最好安装 PythonWin 扩展包。在 PythonWin 中提供了 win32api 函数的封装，以及 MFC 类库的封装，通过 PythonWin 的相关模块可以在 Python 中直接调用 Windows 的 API 函数。PythonWin 的安装步骤如下所示。

(1) 从 PythonWin 官方网站 <https://sourceforge.net/projects/pywin32> 下载 PythonWin 的安装程序 pywin32-210.win32-py2.5.exe。

(2) 双击运行安装程序后如图 1-23 所示。

(3) 单击【下一步】按钮，安装程序将自动搜索 Python 的安装路径，如图 1-24 所示。如果未能找到 Python 的安装路径，则需要检查 Python 的版本是否与 PythonWin 的版本相对应，或者重新安装 Python。

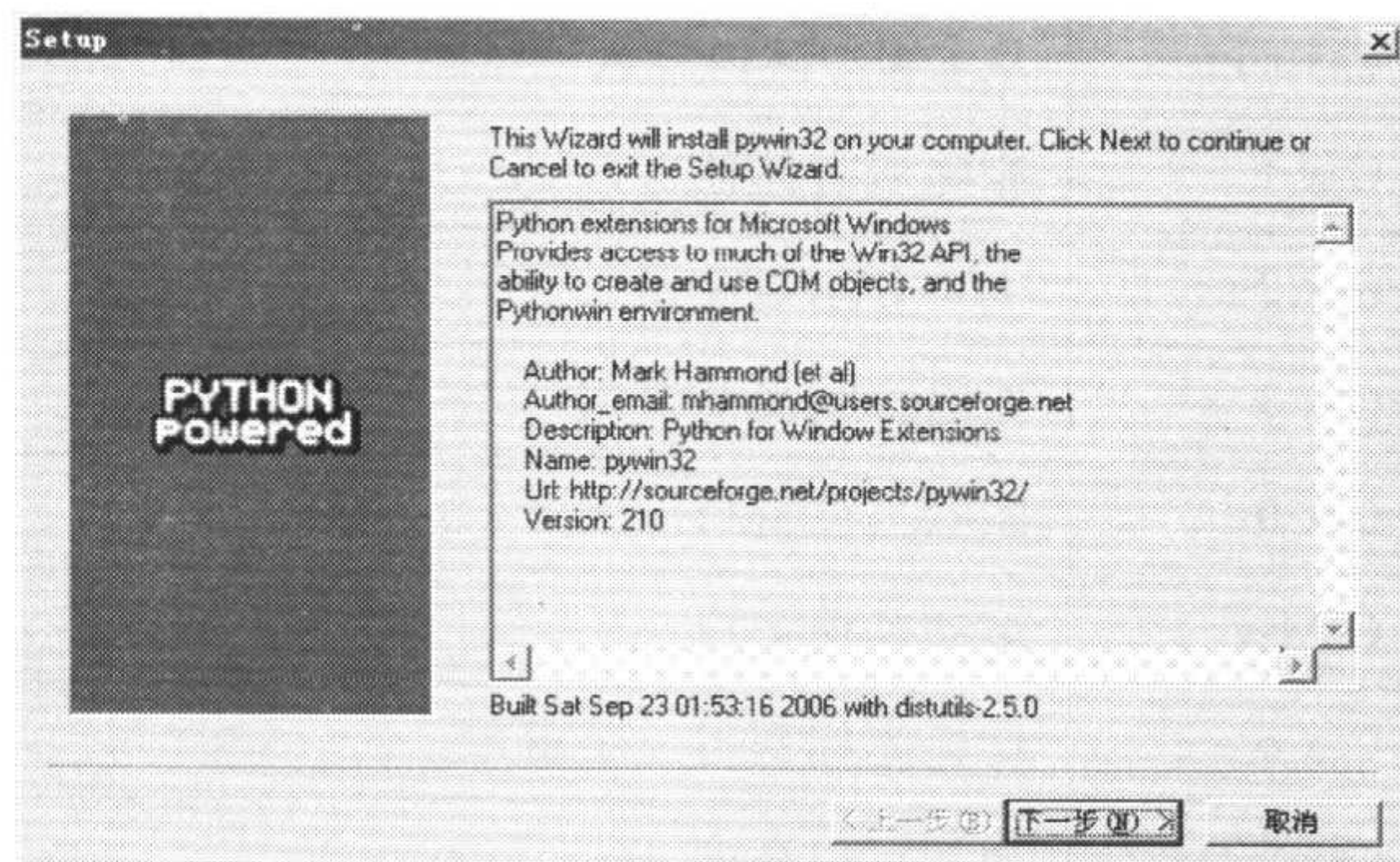


图 1-23 PythonWin 安装程序

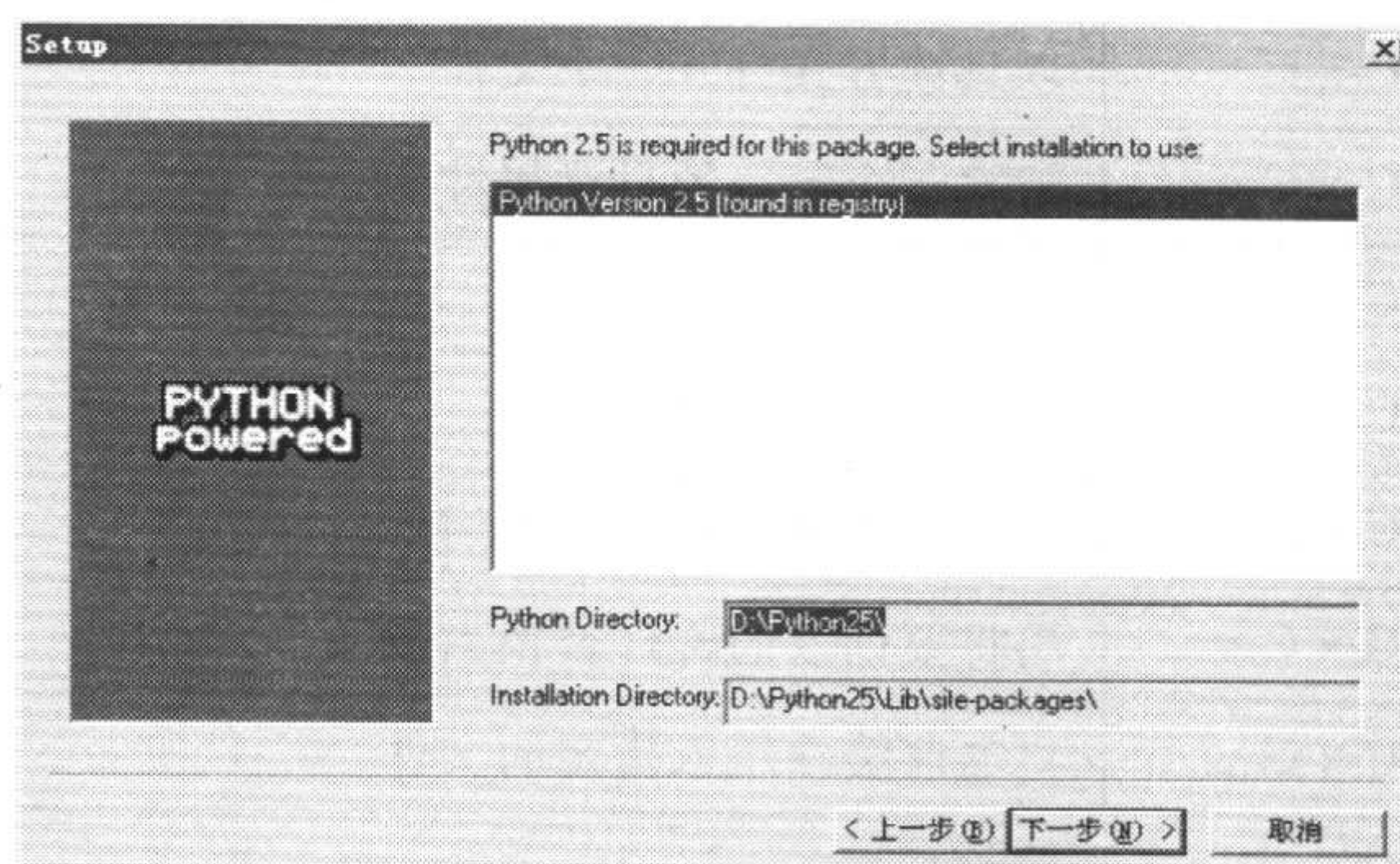


图 1-24 Python 安装路径

(4) 单击【下一步】按钮，进入确认安装界面，如图 1-25 所示。

(5) 单击【下一步】按钮，PythonWin 的安装程序将开始复制安装文件。当文件复制完

成后，将出现如图 1-26 所示的界面。单击【完成】按钮，即可完成 PythonWin 的安装。

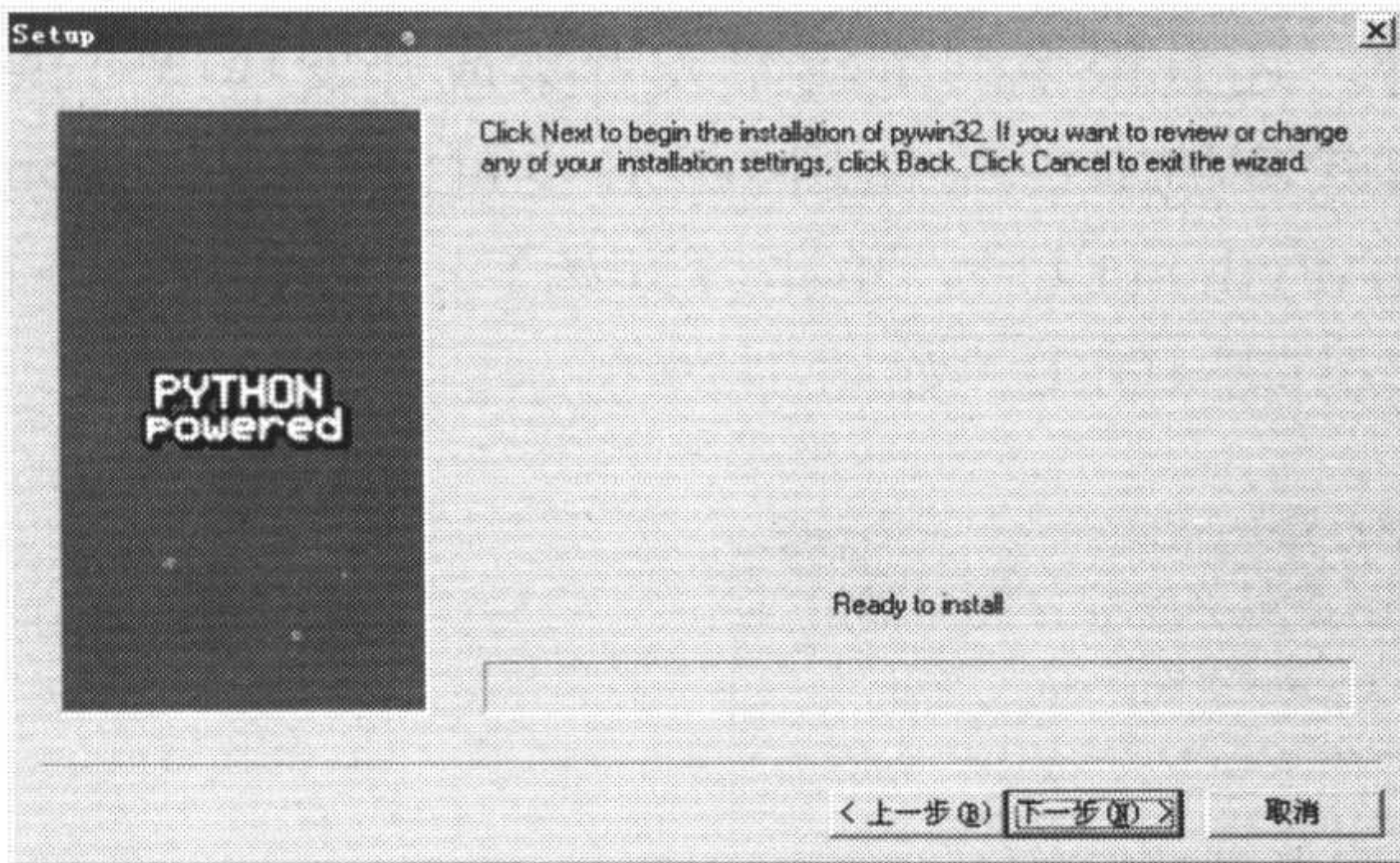


图 1-25 确认安装

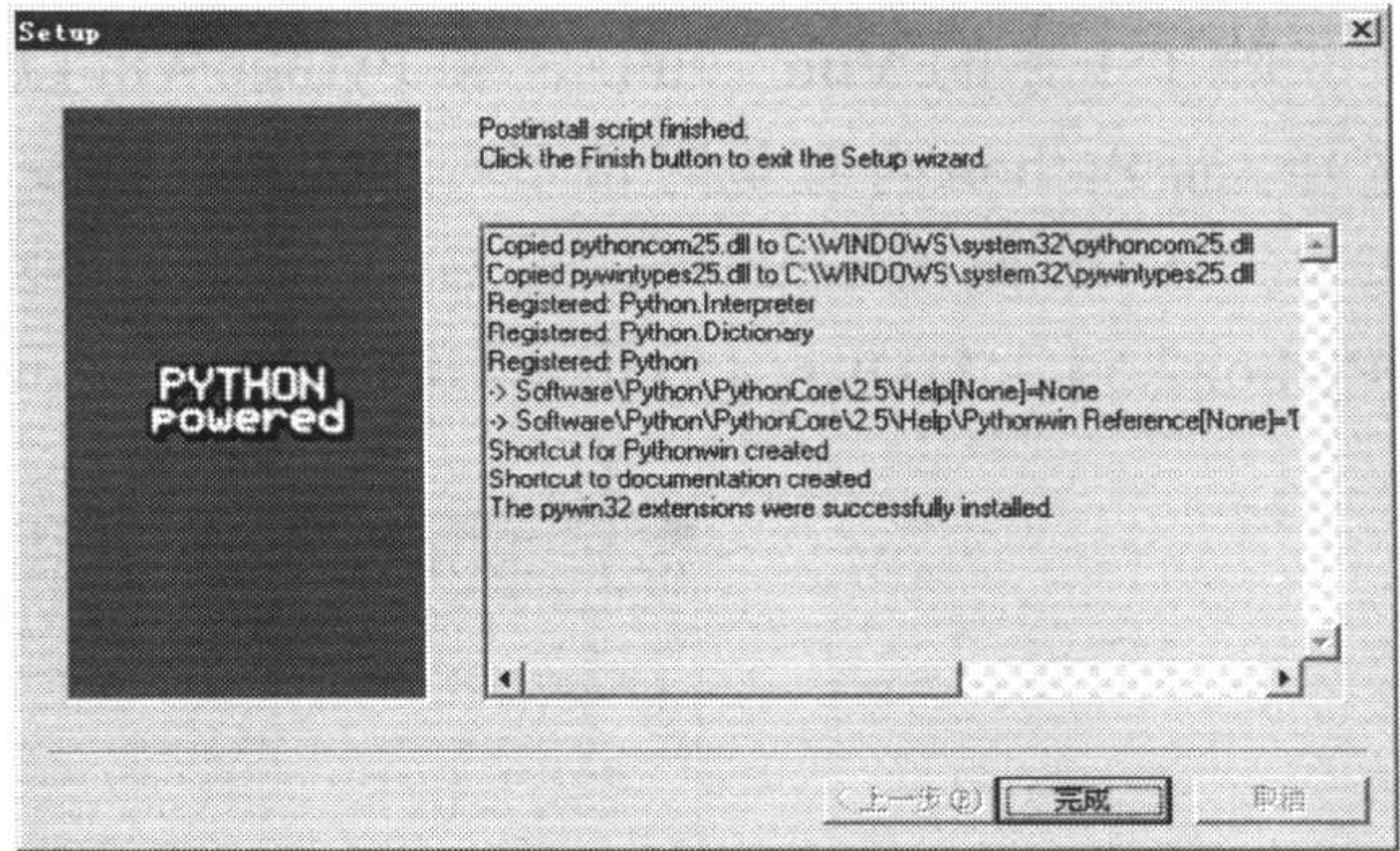


图 1-26 完成安装

2. PythonWin 编辑器简介

PythonWin 提供了一个简单实用的编辑器。在 PythonWin 中不仅可以交互式地运行 Python，还可以编写 Python 脚本。单击【开始】|【所有程序】|【Python2.5】|【PythonWin】命令，将打开 PythonWin 集成环境，如图 1-27 所示。PythonWin 将自动打开 Python 的交互式命令行窗口。单击【File】|【New】命令，可以新建 Python 脚本。

PythonWin 中也提供了自动补全的功能，例如当导入模块后，在模块名后输入“.”，PythonWin 将弹出一个列表窗口，使用方向键上和和下可以选择列表中的项目，按下 Tab 键可以补全，如图 1-28 所示。

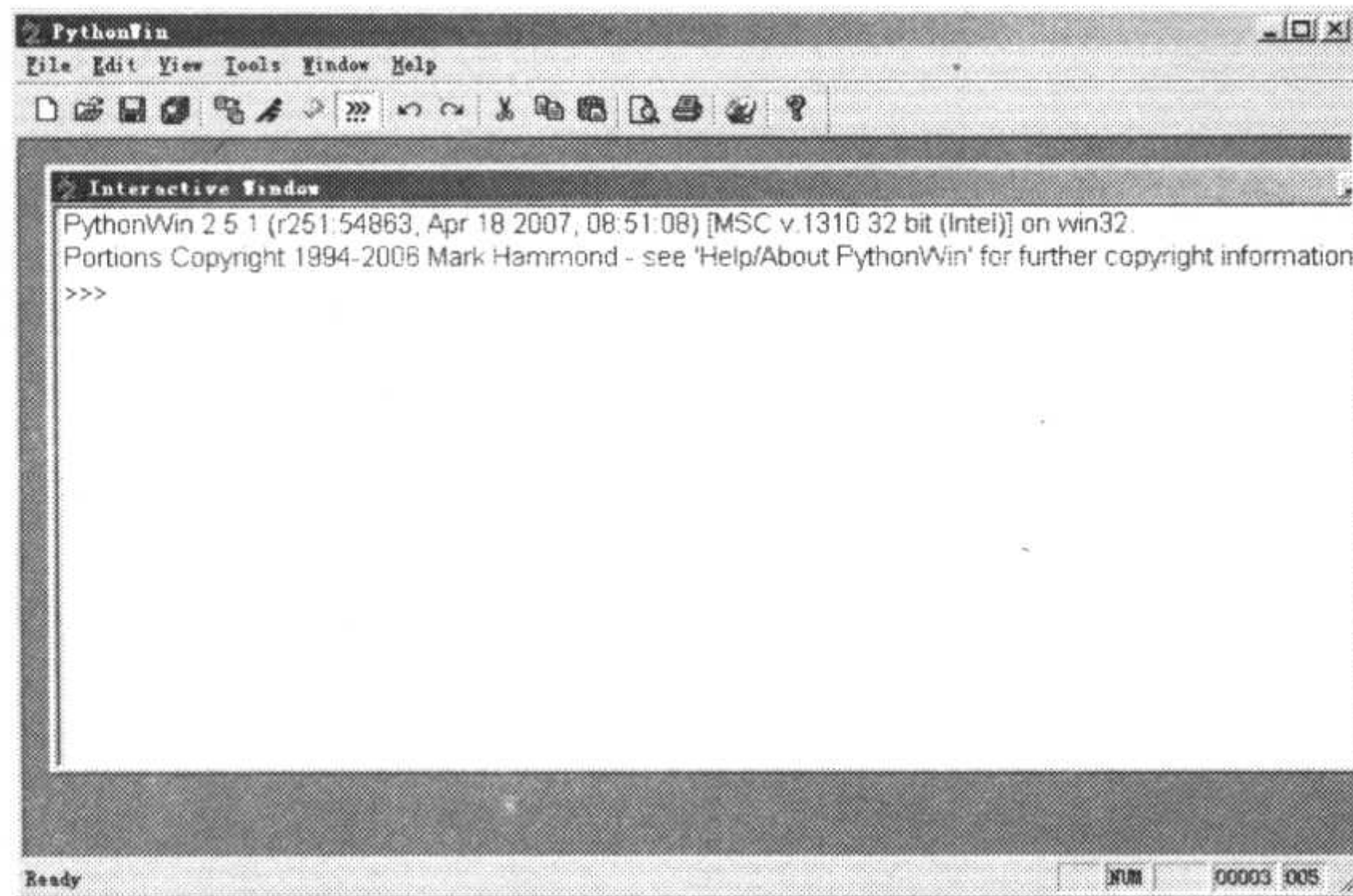


图 1-27 PythonWin 集成环境

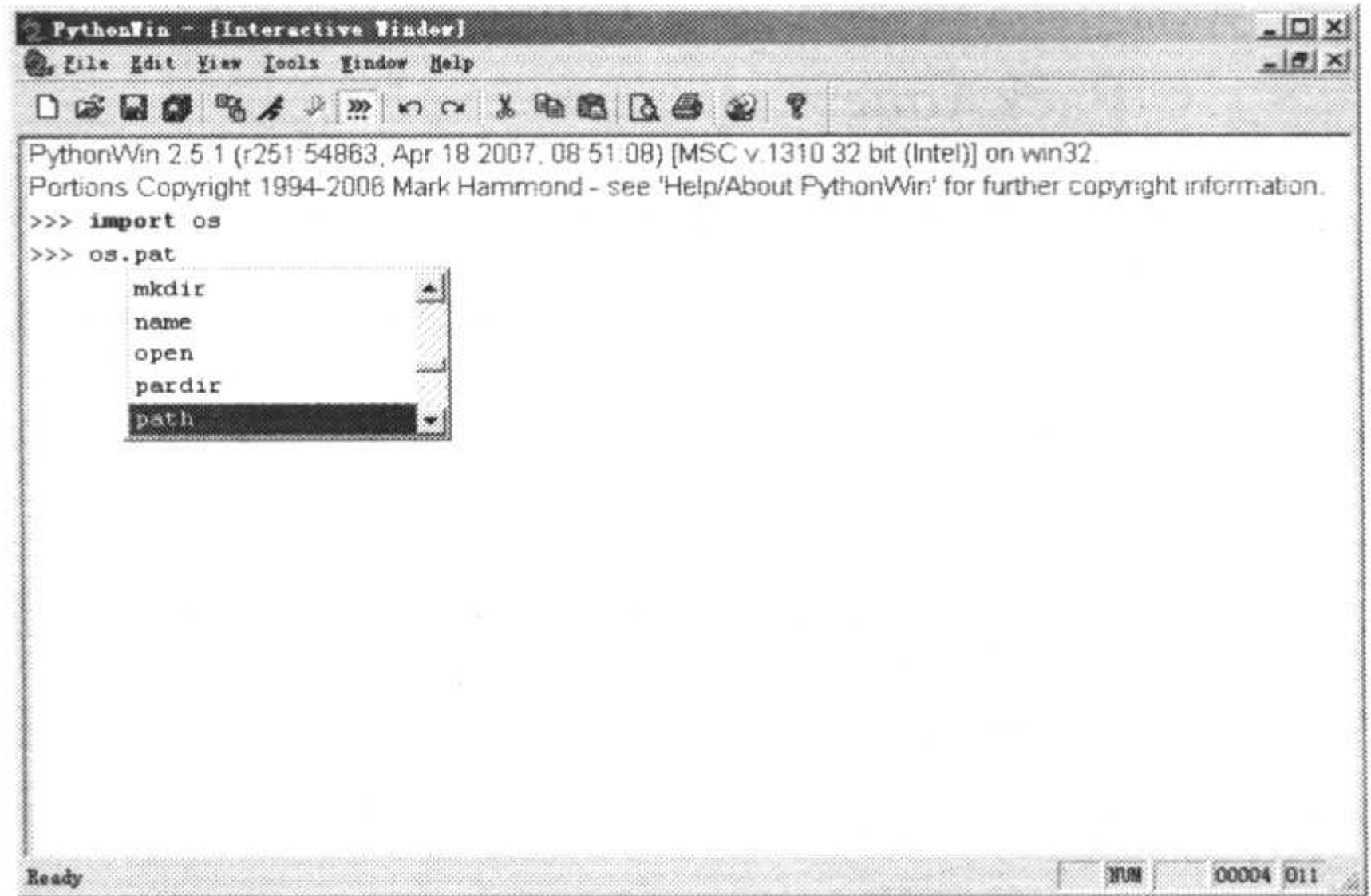


图 1-28 模块函数自动补全

如果所输入的变量已经输入过，则可以按“Alt+/”键来自动补全。另外，在 Python 的交互式命令行下可以按 Ctrl 键加向上的方向键，或者向下的方向键，可以重复输入前面所输入的命令。

1.4.7 其他的 Python 开发环境

除了上述的 Vim、Emacs 和 PythonWin 以外，还有很多的编辑器或者集成开发环境可供选择。而 Python 自身也附带了一个图形化的交互式环境 IDLE，使用 IDLE 也可以编辑 Python 脚本。IDLE 使用 Tkinter 创建 GUI 界面。安装好 Python 以后，单击【开始】|【所有程序】|【Python2.5】|【IDLE (Python GUI)】命令，可以打开 IDLE，如图 1-29 所示。



图 1-29 IDLE GUI

还有一些可以作为 Python 集成开发环境的软件如下所示。

1. BlackAdder

BlackAdder 可以运行在 Windows 和 Linux 系统中。BlackAdder 为用户提供了个人版和专业版，分别面向个人用户和商业用户。BlackAdder 的官方网站为：<http://www.thekompany.com/products/blackadder>。

2. Wing IDE

Wing IDE 是使用 python 编写的，可以运行在 Windows 和 Linux 系统中。Wing IDE 提供一个源码分析器、浏览器，以及文本编辑器和调试器。Wing IDE 为共享软件，可以从其网站 <http://www.archaeopteryx.com/wingide> 下载试用版。

3. Komodo

Komodo 是 ActiveState 提供的一个 Python IDE。ActiveState 还有自己的 Windows 版本的 Python。Komodo 可以运行在 Windows 和 Linux 系统中，它不仅为 Python 提供了集成环境，还为 Perl、PHP、Tcl 和 HTML 等提供了集成开发环境。Komodo 为共享软件，可以从其网站 http://www.activestate.com/products/komodo_ide 下载试用版。

4. Boa Constructor

Boa 是使用 Python 和 wxPython 编写的跨平台的 Python IDE。它提供可视化的编程和操作框架，能方便地进行程序的设计。Boa 提供了对象浏览器，并提供各种资源的视图。在 Boa 中还包含一个 html 文档生成器，以及调试器和完整的帮助系统。Boa 还提供对 zope 的支持。Boa 可以从其官方网站下载 <http://boa-constructor.sourceforge.net>。Boa 是免费的，在安装 Boa 之前要先安装合适版本的 Python 和 wxPython。

5. PyDev

PyDev 是 Eclipse 中的 Python 开发插件。Eclipse 是著名的跨平台的自由集成开发环境，主要用来进行 Java 语言开发。Eclipse 本身只是一个框架平台，但是众多插件的支持使得它拥有其他功能相对固定的 IDE 软件很难具有的灵活性。许多软件开发商以 Eclipse 为框架开发自己的 IDE。可以从 PyDev 的官方网站 <http://pydev.sourceforge.net> 下载 PyDev。在安装 PyDev 之前还应该安装 Eclipse。

6. Eric3

Eric3 是一个功能强大的 Python IDE，它基于 QScintilla 编辑器组件，使用 Python 和 PyQt 编写。Eric3 中集成项目管理工具，可以生成类 UML 的图表，还包含一个功能强大的 Python 调试器。Eric3 可以从其官方网站 <http://www.die-offenbachs.de/detlev/eric3.html> 下载。

7. DrPython

DrPython 是一个跨平台的高可配置的程序开发环境，使用 Python 编写。DrPython 基于 wxPython 和 Scintilla 库。DrPython 可以从其官方网站 <http://drpython.sourceforge.net> 下载。

8. SciTE

SciTE 是基于 SCIntilla 和 GTK+ 开发的，可以运行在 Windows 和 Linux 系统中。SciTE 支持语法高亮、代码折叠等，并可以将代码导出为 HTML、RTF 和 PDF。SciTE 可以从其官方网站 <http://scintilla.sourceforge.net> 下载。

1.5 运行 Python 脚本

在 Windows 下有多种方式可以运行 Python 脚本，也可以直接在 Python 的交互式命令行下一句一句地编写运行 Python 脚本。当代码很多的时候，则应该在文本编辑器中将代码编辑好，然后再运行。

1.5.1 第一个 Python 程序——“Hello, Python!”

如果读者学习过 C 语言，应该使用 C 语言编写过第一个“Hello, World”程序。使用 Python 编写这样的程序仅需要一行代码，如下所示。

```
print 'Hello, Python!'
```

打开文本编辑器，输入上述代码，然后将其保存为“HelloPython.py”。在 Windows 下可

以通过直接双击“HelloPython.py”运行脚本。如果双击运行“HelloPython.py”，看到出现一个命令行窗口，然后又关闭，由于很快，可能看不到输出内容。为了能看到脚本的输入内容，可以单击【开始】|【运行】命令，在【打开】文本框中输入“cmd”命令，打开 Windows 的命令行，如图 1-30 所示。然后进入“HelloPython.py”所在的目录，如“E:\Python”，在命令行提示符下输入“HelloPython.py”或者“python HelloPython.py”，然后按回车键即可运行“HelloPython.py”脚本，如图 1-31 所示。

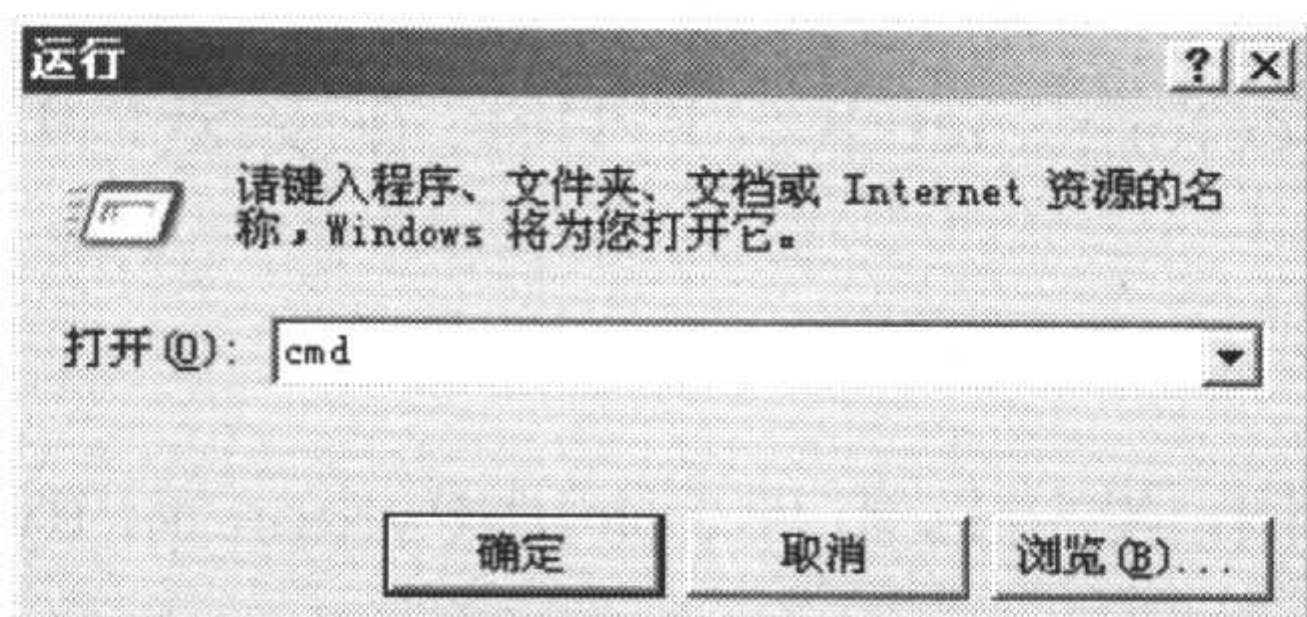


图 1-30 打开 Windows 的命令行

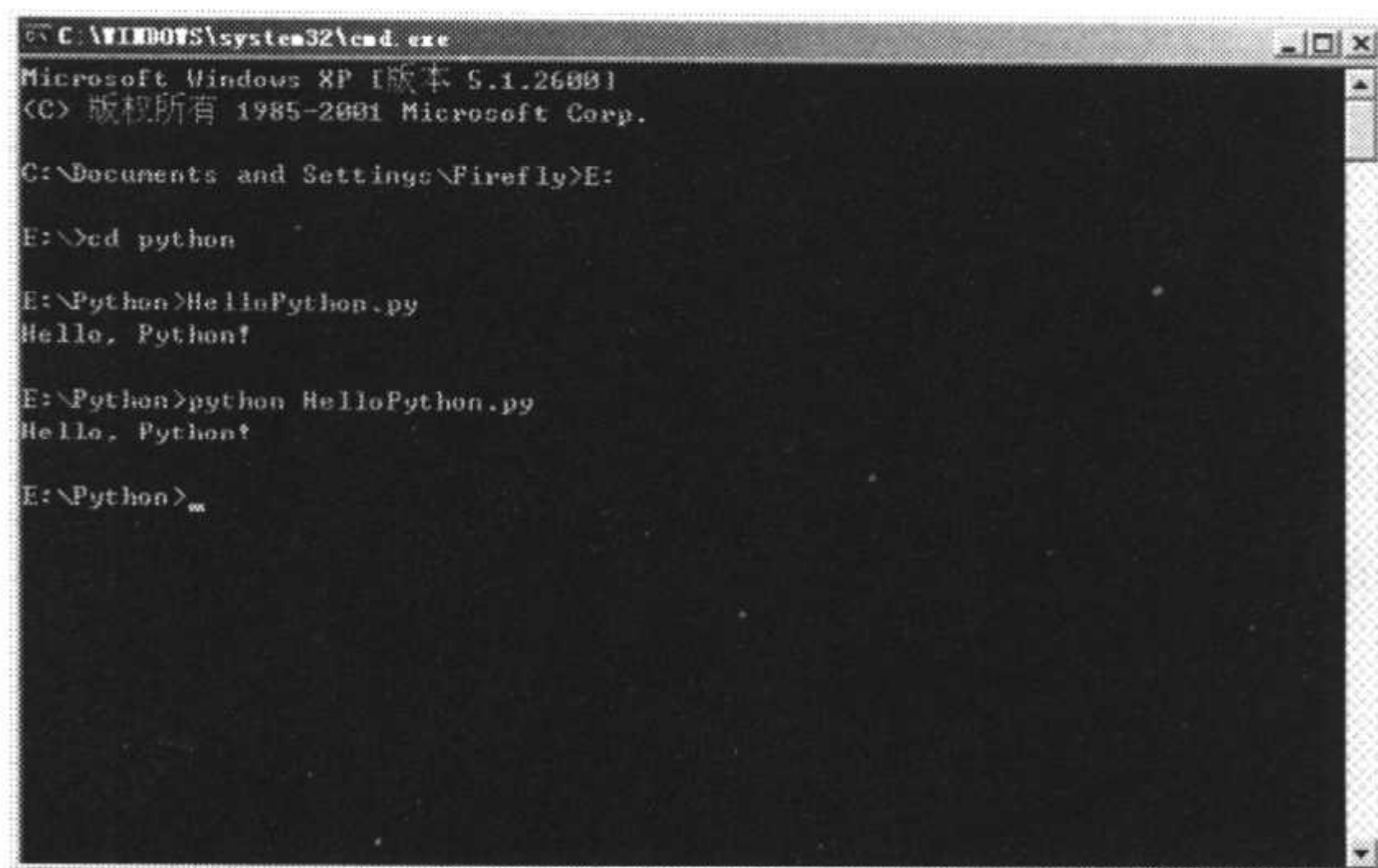


图 1-31 运行“HelloPython.py”脚本

1.5.2 在 Python 交互式命令行中运行脚本

Python 还提供了交互式命令行，可以一边输入脚本，一边运行脚本。通过单击【开始】|【所有程序】|【Python2.5】|【Python (command line)】命令，可以打开 Python 的交互式命令行，在命令行中输入“print 'Hello,Python!'”，然后按【Enter】键，如图 1-32 所示。

如果安装了 PythonWin，可以使用 PythonWin 中的 Python 交互式命令行，可以方便地粘贴、复制代码，使用 PythonWin 提供的自动补全功能，方便程序输入。单击【开始】|【所有程序】|【Python2.5】|【PythonWin】命令，打开 PythonWin。单击【File】|【Open】命令，打开“HelloPython.py”脚本，如图 1-33 所示。

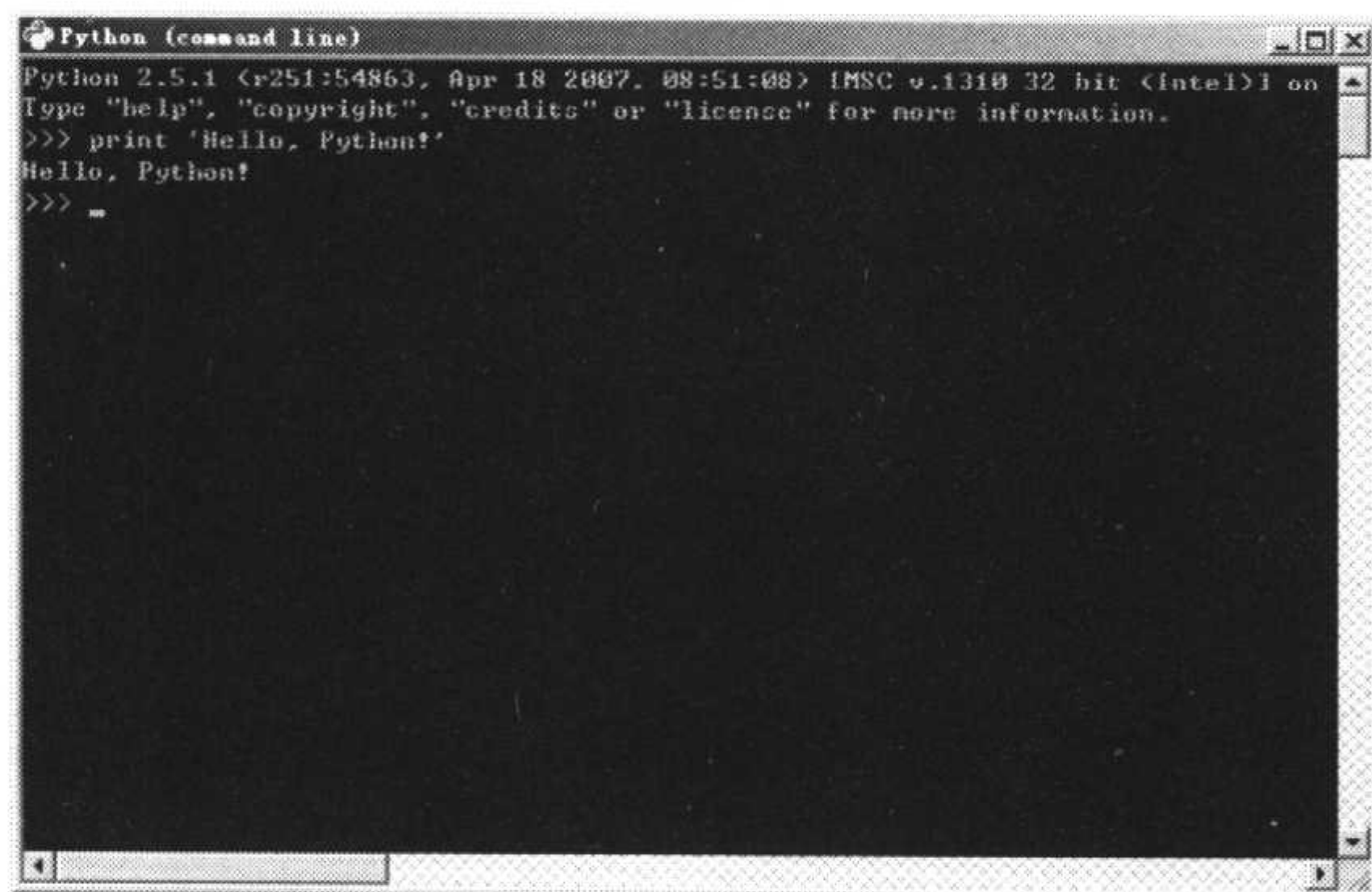


图 1-32 Python 交互式命令行

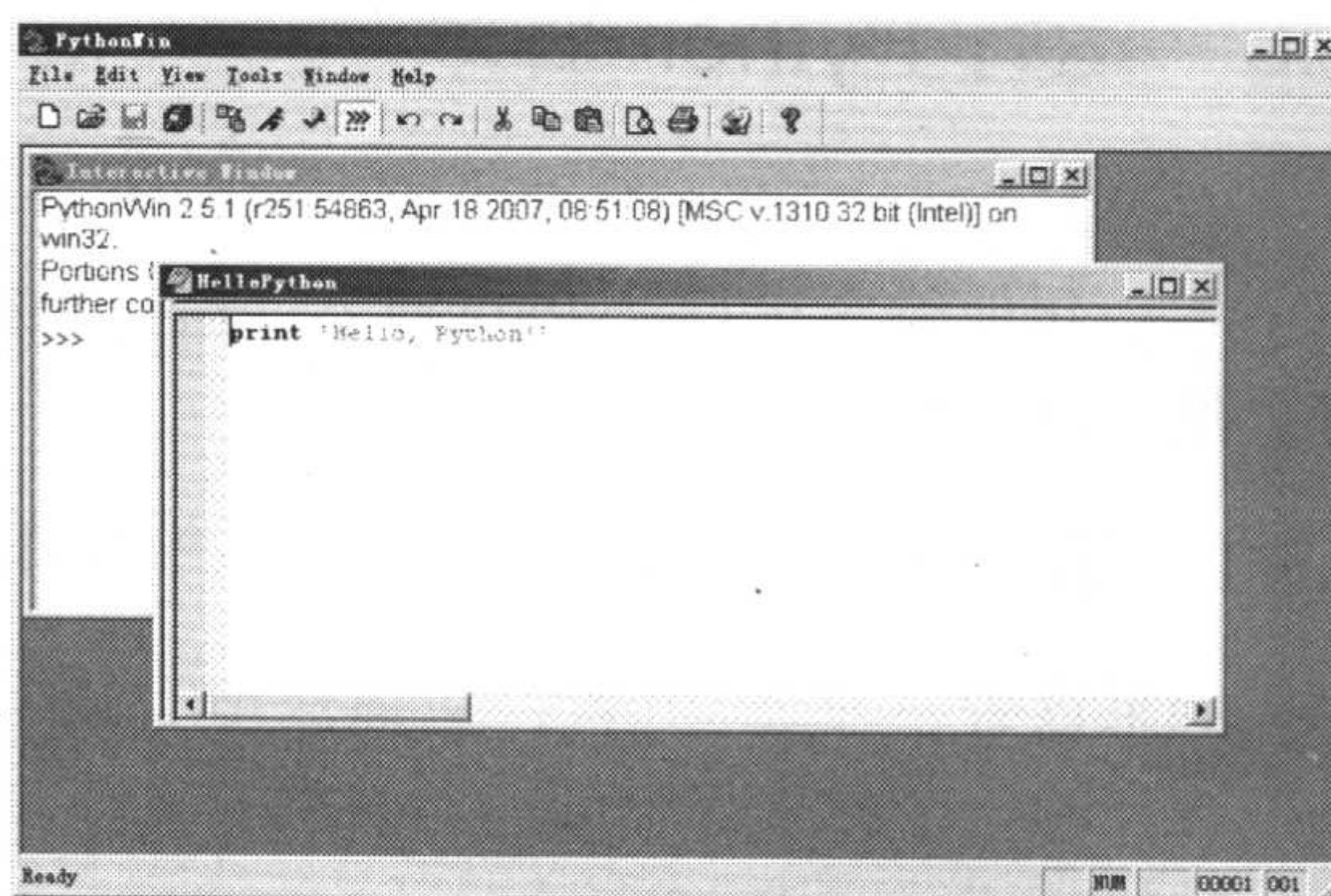



图 1-33 在 PythonWin 中打开“HelloPython.py”脚本

第1章 Python 概述

单击  图标，弹出如图 1-34 所示的对话框。单击【OK】按钮，即可运行“HelloPython.py”脚本，此时 PythonWin 的交互式窗口如图 1-35 所示。

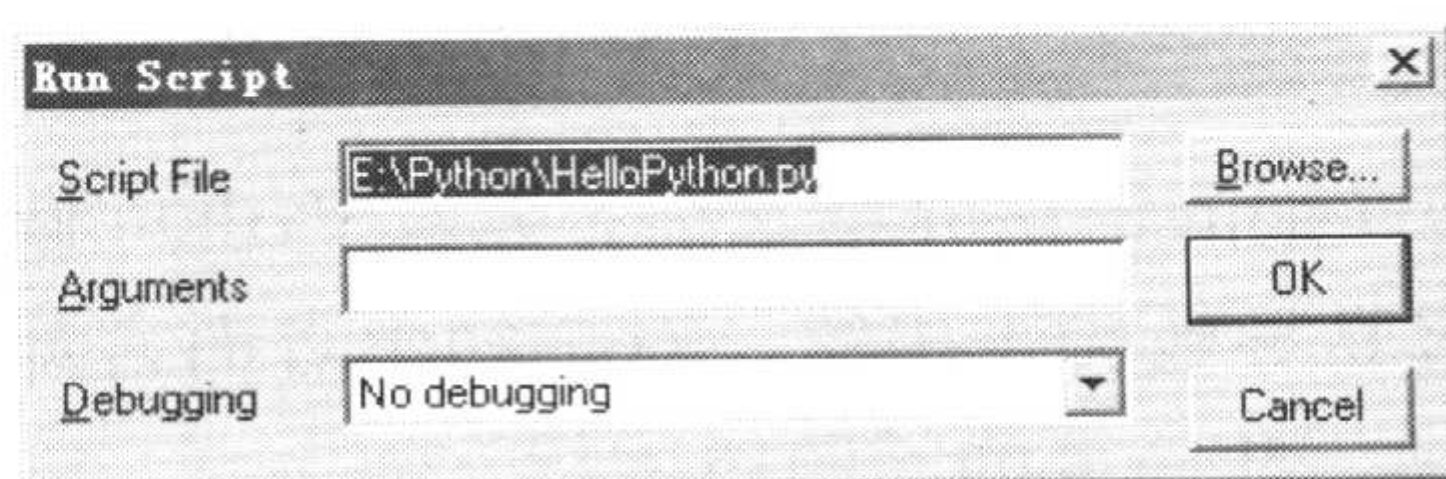


图 1-34 运行脚本

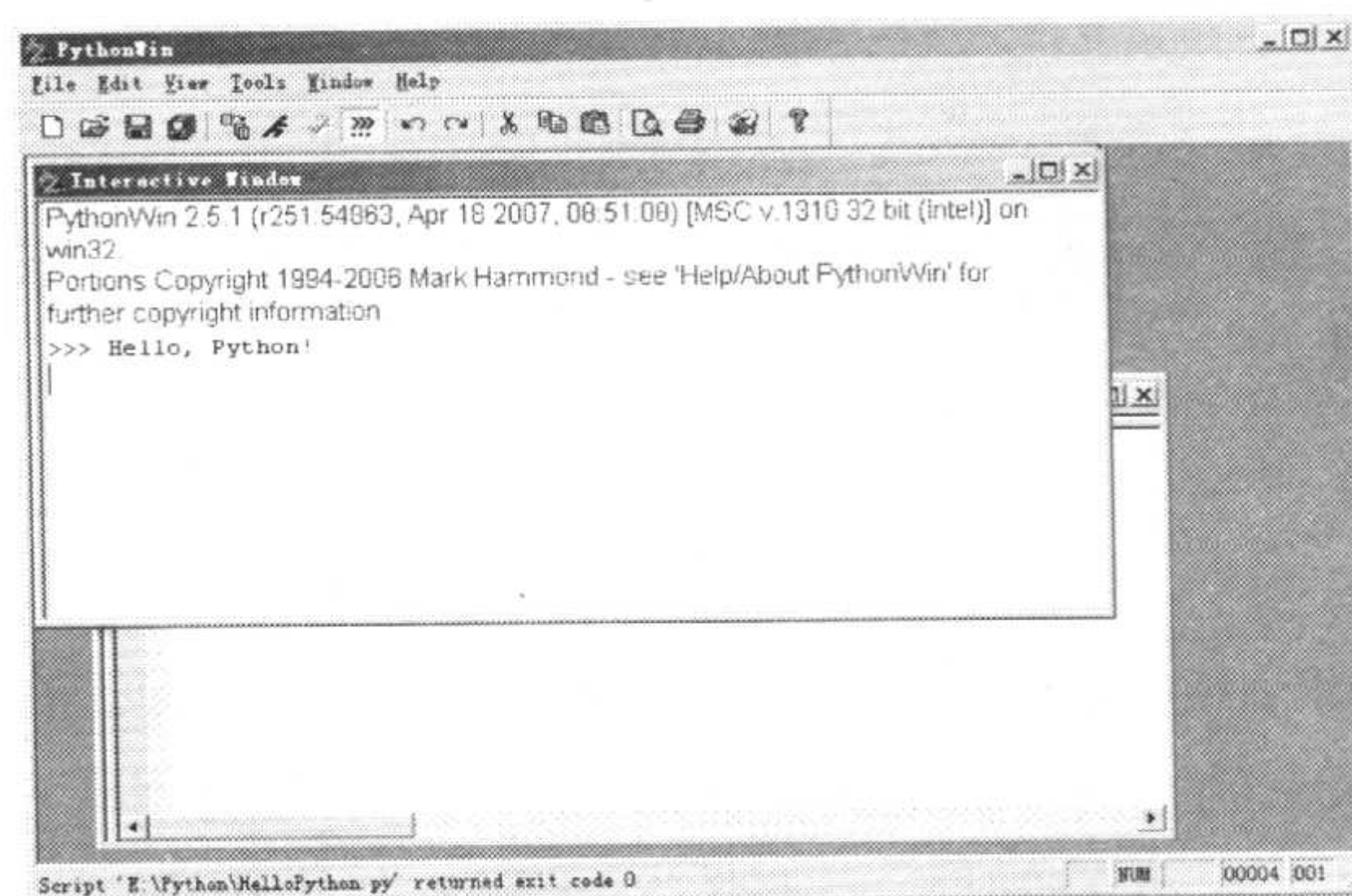


图 1-35 脚本输出

CHAPTER 2

第 2 章 Python 起步

Python 语法简单，非常容易学习、掌握。在开始详细的语法介绍之前，先对 Python 做一下整体的介绍。如脚本的结构、中文的使用等一些容易导致脚本出错，对于初学者而言却又不容易解决的问题，以避免初学者在使用 Python 时被一些简单的问题所困扰。

2.1 脚本基本结构

Python 脚本的结构非常清晰，在 Python 中不使用 C 语言中的花括号表示语句块，而使用缩进。代码缩进一般用在函数定义、类的定义以及一些控制语句中。一般来说，行尾的“:”表示代码缩进的开始。如下所示为在判断语句中使用缩进。

```
if a > b:
    print a                # 代码缩进，如果 a>b，则执行 print a
else:
    print b                # 代码缩进，如果 a<=b，则执行 print b
```

如下所示为一个比较复杂的代码缩进的例子。

```
if a > b:
    if a == 1:             # 代码缩进
        print a           # 代码缩进，即缩进嵌套
    else:
        if a == 0:
            print a
        else:
            pass
elif a == b:
    print a,b
else:
    print b
```

需要注意的是，处于同一级的代码缩进，其缩进量要保持一致，如下所示的例子，代码缩进量不一致，将导致错误。

```
if a > b:
    if a == 1:
        print a
    else:
```

```

        if a == 0:
            print a
        else:
            pass
elif a == b:
    print a
    print b
else:
    print b

```

此处代码和下一句的代码缩进不一致，脚本运行错误

在 Python 中注释语句以字符“#”开头，位于“#”之后的语句不被执行。字符“#”仅注释其所在的行。在 Python 中如果进行大段的注释，可以使用 3 个单引号“'''”或者 3 个双引号“"""”包围。如下所示为两种不同方式的注释。

```

'''
三个单引号包围的注释
该段代码判断 a, b 值的大小
并根据不同的情况输出
如果 a 大于 b 则输出 a
如果 a 小于等于 b 则输出 b
'''
if a > b:
    print a
else:
    print b
'''
三个双引号包围的注释
代码判断结束
print a
上边的语句不会被执行
'''

```

判断 a, b 大小
输出 a
输出 b

在 Python 中，一般来说一条语句占用一行。一条语句结束一般不需要使用 C 语言中的“;”，但在 Python 中也可以使用“;”将两条语句写在一行。另外如果缩进语句中只有一条语句，也可以将其写在“:”之后，如下所示。

```

if a > b: print a
else: print b
print a; print b

```

缩进语句写在冒号之后
使用分号将两条语句写在同一行

在 Python 中，单引号和双引号没有区别，都可以用来包围字符串。另外，单引号中的字符串可以包含双引号，双引号中的字符串可以包含单引号，而不需要使用转义字符，如下所示。

```

a = "What's your name"
b = 'I say: "What is your name?"'

```

另外 3 个单引号或者 3 个双引号所包围的字符不仅可以作为注释，还可以作为格式化的字符。当使用 Python 中的“print”输出这些字符时，其格式保持不变，如下所示。

```

a = '''

```

```

这是格式化的字符
    此处的缩进将输出
在这里也可以使用'
或者"
不影响
"""
当然还有三个双引号
"""
'''
b = """
这是三个双引号包围的
格式化    字符
'''
"""

```

在 Python 中如果语句较长，需要分成几行写时可以使用“\”，或者用一对圆括号来将一条语句写成几行，如下所示。

```

# 使用“\”续行
# 需要注意的是“\”之后不能有任何字符
# 不能在“\”之后使用“#”注释
c = a * 2\
    + b\
    - b\
    * 3
# 使用圆括号包围分成多行的语句
# 在语句中可以使用“#”注释
c = ( a *
      b - 1
      + 3
      /
      2)

```

需要说明的是，在 Python 脚本中所有语句中的标点符号都是英文标点符号。在编写 Python 脚本时最好将输入法切换到英文，避免输入中文标点符号导致脚本运行错误。在字符串和注释中可以使用中文标点。

2.2 基本输入/输出

Python 中的基本输入语句是“raw_input”语句。该语句返回所输入的字符串，如果想要获取数字，可以使用“int”函数将字符串转为数字。如下所示的代码在 Python 的交互式命令行中运行。

>>> raw_input('Input your name:')	# 使用 raw_input 提示输入
Input your name:bluebanboom	# bluebanboom 为用户输入
'bluebanboom'	
>>> name = raw_input('Input your name:')	# 将用户输入赋值给 name
Input your name:bluebanboom	
>>> print name	# 输出 name

第2章 Python 起步

```

bluebanboom
>>> year = raw_input('The year:')           # 获取输入
The year:2007
>>> print year
2007
>>> year + 1                                # year 加 1, 这里导致出错, 因为 year 为字符串型
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> int(year) + 1                            # 使用 int 函数将 year 转换成整型
2008

```

上述代码中需要说明的是位于“>>>”命令提示符之后的为用户输入的语句。如果语句前没有“>>>”命令提示符，则表示该语句为 Python 的输出。但是由于使用了“raw_input”语句，因此在“raw_input”语句的提示之后需要用户输入。如果在 PythonWin 的交互式命令行中使用“raw_input”语句，将弹出如图 2-1 所示的对话框。在文本框中输入内容后，单击【OK】按钮即可。

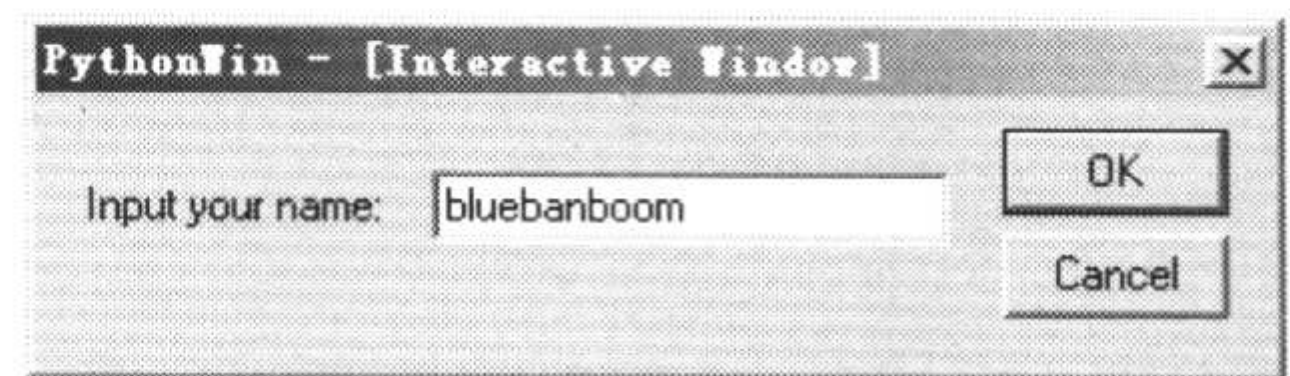


图 2-1 raw_input 对话框

在 Python 中除了 int 函数外，还有以下几个用于类型相互转换的函数。

- float: 将字符串或者整数转换为浮点数。
- str: 将数字转换为字符串。
- chr: 将 ASCII 值转换为 ASCII 字符。
- hex: 将整数转换为十六进制的字符串。
- long: 将字符串转换为长整型。
- oct: 将整数转化为八进制的字符串。
- ord: 将 ASCII 字符转化为 ASCII 值。

Python 中的基本输出语句是“print”语句，在“HelloPython.py”脚本中已经使用了“print”语句。使用“print”语句可以输出 Python 中的所有的数据类型，而不需要事先指定要输出的数据类型。如果自定义了某一新的类型或者类，可以通过重载“_repr_”让“print”语句支持。如下所示的代码在 Python 的交互式命令行中运行。

```

>>> a = 0
>>> print a                                # 输出整型
0                                           # 输出内容
>>> b = 1
>>> print a+b                             # 输出表达式的值
1
>>> print b                                # 输出 b 的值
1
>>> s = 'Hello'                            # 定义字符串
>>> print s                                # 输出字符串

```

```

Hello
>>> l = [1, 2, 3]           # 定义列表
>>> print l                 # 输出列表
[1, 2, 3]
>>> t = ('a', 'b', 'c')    # 定义元组
>>> print t                 # 输出元组
('a', 'b', 'c')
>>> print l,t               # 同时输出列表和元组
[1, 2, 3] ('a', 'b', 'c')
>>> print l, '\n', t       # 使用换行符
[1, 2, 3]
('a', 'b', 'c')
>>> for i in t:             # 循环输出
... print i
...                          # 在空缩进处按一下回车键即表示缩进结束
a
b
c

```

在上述代码中使用了缩进，在交互式命令行下，以“...”表示缩进开始，如果使用的是 Python 自带的交互式命令行，需要注意进行缩进，而如果在 PythonWin 的交互式命令行中则不需要，因为 PythonWin 提供了自动缩进。如果缩进的语句已经结束，只需按一下回车键，表示缩进结束。

2.3 在 Python 中使用中文

在 Python 中可以使用中文，但需要对中文进行处理。在 Python 中，显示中文主要是字符编码的问题，如果处理不好将导致乱码。在计算机中，字符是以数字来表示的。字符通过字符编码将其转化为数字，以使计算机能够对其识别。

最早的时候，计算机仅支持英文字符，也就是 ASCII 字符。ASCII 字符包含大小写字母以及一些标点和其他的字符，用一个字节表示。但是对于中文字符来说，使用一个字节显然不能满足需要。为了能够在计算机中表示所有的中文字符，中文编码采用两个字节表示。如果中文编码和 ASCII 混在一起，就会导致解码错误，而产生乱码。采用两个字节的中文编码标准有 GB2312、GBK、BIG5。为了将各种不同的语言都包含在统一的字符集中，满足国际间的信息交流，国际上制定了 UNICODE 字符集。UNICODE 字符集包含世界上所有语言字符，这些字符具有唯一的编号。通过使用 UNICODE 字符集可以满足跨语言的文字处理，有效地避免乱码的产生。

在 Python 中可以在各种编码间相互转换。如果在交互式命令中使用中文，可以不做处理，一般不会出现乱码。如果在“.py”文件中使用了中文，则需要在文件的第一行使用如下语句指定字符编码集。

```
# -*- coding:utf-8 -*-
```


其中 utf-8 表示使用 utf-8 编码，也就是 UNICODE 字符集。使用上述语句，仅指明脚本中包含非 ASCII 字符，而并未将字符编码转换为 utf-8 编码。如果要将字符编码改为 utf-8，则需要在保存的时候选择保存为 utf-8 的格式。在记事本中可以通过选择【文件】|【另存为】命令，在弹出的“另存为”对话框中的【编码】列表框中选择“UTF-8”，如图 2-2 所示。

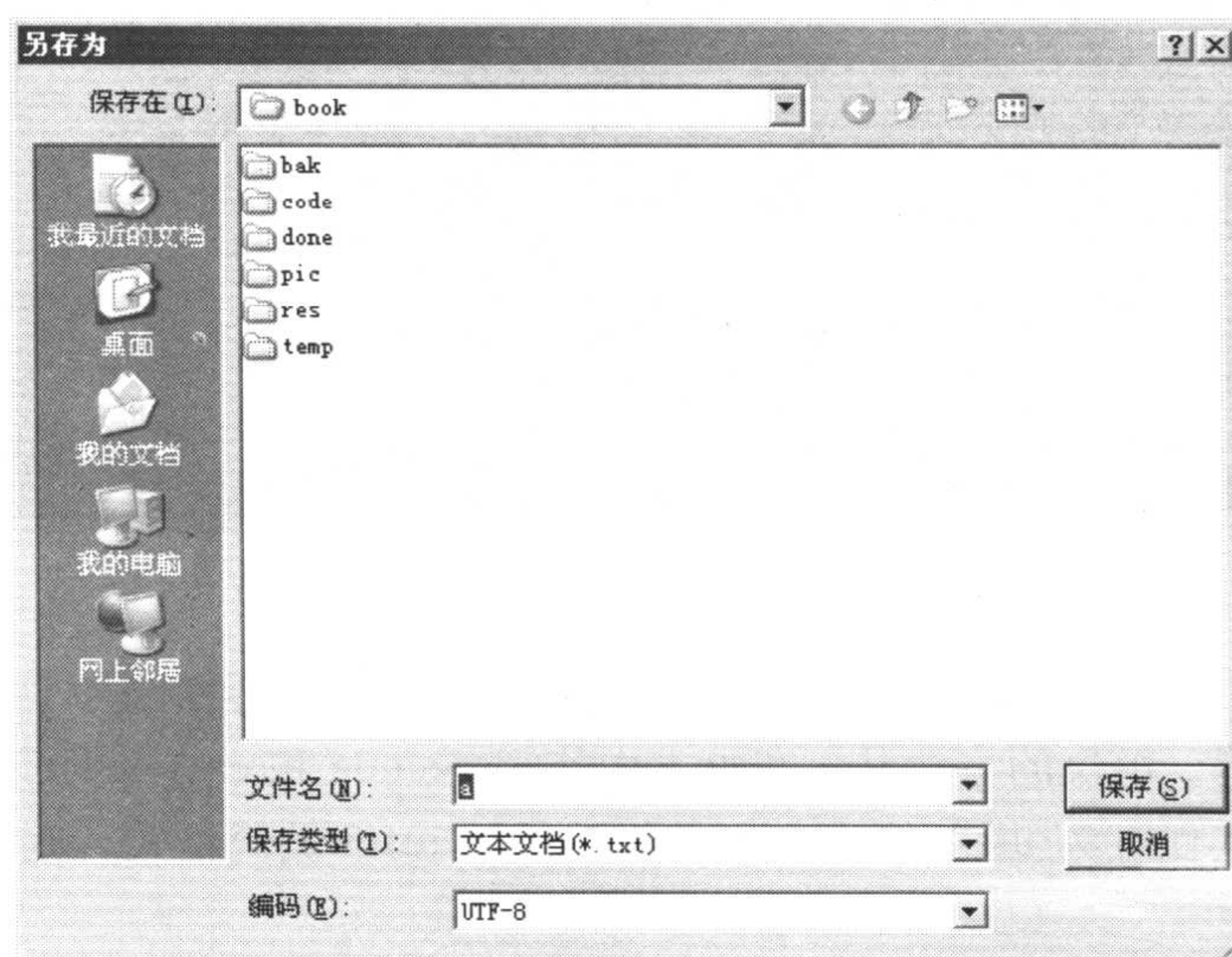


图 2-2 选择文件编码

在 Vim 中可以使用如下命令设置文件的编码。

```
:set fileencoding=utf-8
```

需要注意的是，Vim 在有些情况下选择 utf-8 编码后容易导致乱码，这时，可以通过记事本重新保存为 utf-8 格式。

除了使用 utf-8 编码以外，还可以使用 cp936、gb2312、iso-8859-1 等来指定字符编码，从而在 Python 脚本中使用中文。需要注意的是，如果要在命令行界面中输出中文字符，则需要设定编码为 cp936。例如如下所示的脚本，将字符编码设定为 utf-8，并且保存为 utf-8 编码格式，运行脚本后输出如图 2-3 所示。

```
# -*- coding:utf-8 -*-
# file: Chinese.py
#
chinese = '''
在 Python 中使用中文
需要注意字符编码的问题
可以使用的字符编码有如下几种：
utf-8、cp936、gb2312、iso-8859-1。
'''
print chinese
```




图 2-3 命令行乱码

这是因为在 Windows 的命令行中采用的是 cp936 编码，而在上述脚本中采用的是 utf-8 编码，因此导致乱码。解决的方法是，在脚本中使用 decode 和 encode 函数对字符重新解码、编码，或者，不将其保存为 utf-8 格式。如果仍然采用 utf-8 编码格式，则可以将脚本修改为如下所示。

```
# -*- coding:utf-8 -*-
# file: Chinese.py
#
chinese = '''
在 Python 中使用中文
需要注意字符编码的问题
可以使用的字符编码有如下几种：
utf-8、cp936、gb2312、iso-8859-1。
'''
print chinese.decode('utf-8').encode('cp936')
```

2.4 把 Python 当作计算器

由于 Python 具有交互式的命令行，在交互式命令行下可以使用 Python 完成基本的数学计算。如下代码可在 Python 交互式命令行中运行。

```
>>> 3*5/2          # 结果为整数
7
>>> 3.0*5.0/2      # 结果为小数
7.5
>>> 3.0+5.0/2
5.5
>>> (3.0+5.0)/2    # 使用括号改变运算符优先级
```



```
4.0
>>> 2**3          # 求 2 的 3 次方
8
>>> 2**8          # 求 2 的 8 次方
256
```

另外在 Python 的 math 模块中提供了一些基本的数学函数, 其中主要的函数有以下几个。

- `sin(x)`: 求 x 的正弦。
- `cos(x)`: 求 x 的余弦。
- `asin(x)`: 求 x 的反正弦。
- `acos(x)`: 求 x 的反余弦。
- `tan(x)`: 求 x 的正切。
- `atan(x)`: 求 x 的反正切。
- `hypot(x, y)`: 求直角三角形的斜边长度。
- `fmod(x, y)`: 求 x/y 的余数。
- `ceil(x)`: 取不小于 x 的最小整数。
- `floor(x)`: 取不大于 x 的最大整数。
- `fabs(x)`: 求绝对值。
- `exp(x)`: 求 e 的 x 次幂。
- `pow(x, y)`: 求 x 的 y 次幂。
- `log10(x)`: 求 x 的以 10 为底的对数。
- `sqrt(x)`: 求 x 的平方根。
- `pi`: π 的值。

如下所示的代码使用 Python 中的 math 模块进行数学计算。

```
>>> import math
>>> math.cos(0.5)
0.87758256189037276
>>> math.sin(math.pi)
1.2246063538223773e-016
>>> math.sin(60)
-0.30481062110221668
>>> math.tan(1)
1.5574077246549023
>>> math.sqrt(9)
3.0
>>> math.log10(2)
0.3010299956639812
>>> math.log10(100)
2.0
>>> math.asin(0.5)
```

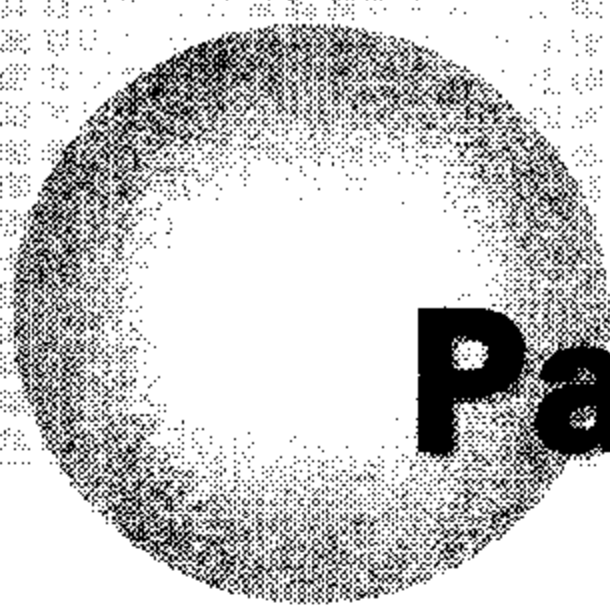
```
0.52359877559829893
>>> math.pow(2,8)
256.0
```

另外在 Vim 中也可以使用 Python 的表达式进行数学计算。例如输入如下命令可以在 Vim 的命令缓冲区中显示 3 的 5 次幂。

```
:py print 3**5
```

可以通过如下命令导入 math 模块，并使用其中的函数进行计算。

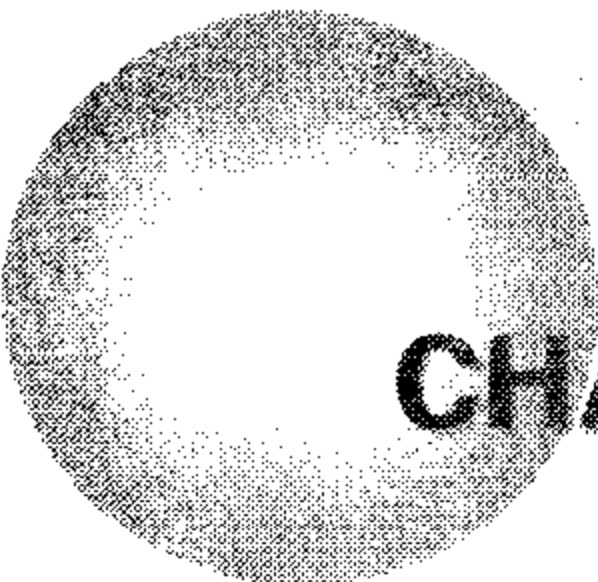
```
:py import math
:py print math.sin(1)
```

Part 2

第二篇

Python 语法



第 3 章 Python 数据类型与基本语句

对于一般的 Python 脚本只要使用 Python 的内置数据类型就可以完成绝大多数工作。使用 Python 编写脚本基本上不必考虑自己重新定义数据类型或是数据结构。3 种基本的语句已经可以完成程序流程的控制。这一章将介绍 Python 的基本数据类型和基本语句。

3.1 Python 数据类型——数字

数据类型是程序的基础。程序设计的本质就是对数据进行处理。在 Python 中设计良好的数据类型，以及其对内置函数的支持，使得 Python 脚本中数据的处理变得十分简单。大多数情况下只要使用内置数据类型就足够了。

数字是最基本的数据类型，任何编程语言都提供了对数字的支持。在 Python 中可以使用任意大的数字，而不用担心溢出，而且 Python 的 math 模块提供了常用的数学函数，如开方、三角函数等，在上一章中已经做过简单的介绍。

3.1.1 基本类型

数字在 Python 中有 4 种类型——整数、长整数、浮点数和复数，如表 3-1 所示。

表 3-1 Python 中数字类型

类 型	描 述
整数	一般意义上的数，包含八进制（以数字 0 开头）及十六进制（以 0x 开头），如 2007，-2007，07（八进制），0xAB（十六进制）等
长整数	无限大小的数，在其结尾添加小写字母 l 或者大写字母 L，如 2007000000000000000L
浮点数	小数或者用 E 或 e 表示的幂，如 2.7，1234e+10，1.5E-10
复数	复数的虚部以字母 j 或者 J 结尾，如 1+2j，2.2+2.0J

作为动态类型的语言，在 Python 中使用数字无需事先声明其类型，如下所示。

```
>>> a = 1                # 将 a 赋值为 1，整数
>>> b = 12.5             # 将 b 赋值为 2，浮点数
>>> a + b                 # 计算 a + b
```

```
13.5
>>> c = 2007000000000000000L      # 长整数
>>> c
2007000000000000000L
>>> d = 20070000000000000001      # 长整数
>>> d
2007000000000000000L
>>> d - c                          # 计算 d - c
1806300000000000000L
>>> d + b
2.007e+017                          # 浮点数
>>> 2.30 - 1.30
0.999999999999999978                # 结果并不为 1.00，由浮点数的精度所导致
>>> 2.3 - 1
1.29999999999999998                # 同样，这里的整型数字 1 被转换成浮点数进行运算
>>> 07 + 05                          # 八进制
12                                  # 输出为十进制
>>> 0x7 + 0xa                        # 十六进制
17                                  # 输出为十进制
>>> print '%o' % ( 07 + 05)          # 输出为八进制
14
>>> print '%x' % ( 0x7 + 0x5)        # 输出为十六进制
c
>>> m = 9 + 3j                      # 复数形式
>>> n = 15 - 2j
>>> m + n                            # 复数运算
(24+1j)
```

3.1.2 运算符

在 Python 中除了基本的算术运算符加、减、乘、除、取余等运算符以外，还有逻辑运算符，如位移运算符、位逻辑运算符等，如表 3-2 所示。

表 3-2 Python 运算符

运 算 符	描 述	运 算 符	描 述
**	乘方运算符		位或运算符
*	乘法运算符	^	位异或运算符
/	除法运算符	&	位与运算符
%	取余运算符	<<	左移运算符
+	加法运算符	>>	右移运算符
-	减法运算符		

在复杂的表达式中往往使用多个运算符。在 Python 中，表达式的计算顺序由运算符的优先级确定。表 3-2 所示的运算符越往下优先级越低。在表达式中先进行运算符优先级高的计算，对于同级运算符从左至右依次计算。

表 3-2 中，乘方运算符的优先级最高；乘法运算符、除法运算符、取余运算符它们的优先级相同，其优先级较乘方运算符次之；加法运算符和除法运算符属同级运算符，它们的优先级次之；剩下的逻辑运算符属于同级运算符，它们的优先级最低。如果要使优先级低的运算符具有高优先级，可以使用括号将表达式括起来，如下所示。

```
>>> 2 ** 5           # 乘方运算，求解 2 的 5 次方
32
>>> 2 ** 0           # 求解 2 的 0 次方
1
>>> 3 * 2             # 乘法运算
6
>>> 4 / 2             # 除法运算
2
>>> 7 / 2
3                     # 结果取整数
>>> 7 % 2             # 取余运算
1
>>> 5 ^ 3             # 位异或，5 的二进制形式为 101，3 的为 011，异或后为 110 即十进制的 6
6
>>> 5 ^ 5
0
>>> 11 | 5            # 位或运算
15
>>> 12 & 12           # 位与运算
12
>>> 2 * 5 ** 2        # 这里先计算 5 ** 2
50
>>> 2 + 3 * 5          # 这里先计算 3 * 5
17
>>> 2 + 5 ^ 5          # 这里先计算 2 + 5
2
>>> 3 + 4 * 5 ** 2 - 20 # 先计算 5 ** 2 = 25，然后计算 4 * 25 = 100，再计算 3 + 100 = 103，最后计算 103 - 20
83
>>> 4 >> 2            # 右移两位相当于除以 4
1
>>> 4 >> 1            # 右移一位相当于除以 2
2
>>> 2 + (3 ^ 5)        # 改变运算顺序，先计算 3 ^ 5
8
>>> (2 + 3) * 5        # 改变运算顺序，先计算 2 + 3
25
```

3.2 Python 数据类型——字符串

Python 中的字符串用于表示和存储文本。字符串通常由单引号（'...'）、双引号（"..."）或者三引号（'''...'''，"""..."""）包围。其中由三引号包围的字符串可以由多行组成。在 Python

中大段的叙述性字符串通常由三引号包围。

3.2.1 字符串概述

字符串中可以包含数字、字母，以及一些控制字符，如换行符、制表符等。字符串的形式一般如下所示。

```
>>> str1 = 'abcd'           # 使用单引号
>>> str2 = "Python"        # 使用双引号
>>> str3 = '123'
>>> str4 = 'a = 1 + 2 ^ 3 * 4'
>>> str5 = 'Can\'t'         # 在字符串中使用转义字符包含一个单引号
>>> str5
"Can't"
>>> str6 = "Can't"         # 使用双引号包含一个单引号
>>> str6
"Can't"
```

如果要在字符中包含控制字符，或者一些在 Python 中表示特殊含义的符号，需要使用转义字符。常见的转义字符如表 3-3 所示。

表 3-3 常见转义字符

转 义 字 符	含 义	转 义 字 符	含 义
\n	换行符	\\	表示\
\t	制表符	\'	表示一个单引号，而不是字符串结束
\r	回车	\"	表示一个双引号，而不是字符串结束

以下实例演示转义字符在字符串中的使用。

```
>>> t = 'Hi,\tPython!'      # 在字符串中加入制表符
>>> print t
Hi, Python!                # 只有使用 print 输出字符串时才会解释字符串中的转义字符
>>> t
'Hi,\tPython!'
>>> t = 'Hi,\nPython!'      # 在字符串中加入换行符
>>> print t
Hi,
Python!
>>> t = 'Hi,\rPython!'      # 在字符串中加入回车，相当于使用换行符
>>> print t
Hi,
Python!
>>> t = 'Hi,\\nPython!'     # 在字符串中加入“\”，而不是用于转义字符
>>> print t
Hi,\nPython!
```

3.2.2 操作字符串

Python 中提供很多对字符串操作的函数，也可以使用“+”、“*”等运算符对字符串进行

操作。其中常用的对字符串的操作如表 3-4 所示。

表 3-4 常用字符串操作

字符串操作	描 述
string.capitalize()	将字符串的第一个字母大写
string.count()	获得字符串中某一子字符串的数目
string.find()	获得字符串中某一子字符串的起始位置
string.isalnum()	检测字符串是否仅包含 0-9A-Za-z
string.isalpha()	检测字符串是否仅包含 0-9A-Za-z
string.isdigit()	检测字符串是否仅包含字母
string.islower()	检测字符串是否均为小写字母
string.isspace()	检测字符串中所有字符是否均为空白字符
string.istitle()	检测字符串中的单词是否为首字母大写
string.isupper()	检测字符串是否均为大写字母
string.join()	连接字符串
string.lower()	将字符串全部转换为小写
string.split()	分割字符串
string.swapcase()	将字符串中大写字母转换为小写，小写字母转换为大写
string.title()	将字符串中的单词首字母大写
string.upper()	将字符串中全部字母转换为大写
len(string)	获取字符串长度

函数的使用如下所示。

```
>>> str = 'hi, python!'
>>> str.capitalize()           # 将字符串的第一个字母大写
'Hi, python!'
>>> str.count('p')             # 获得字符串中“p”的数目
1
>>> str.find('hello')          # 获得字符串中“hello”的起始位置
-1                             # -1 表示未找到
>>> str.find('p')              # 获得字符串中“p”的起始位置
4                             # 从 0 开始也就是字符串中第 5 个字符
>>> str.isalnum()              # 检测字符串是否仅包含 0-9A-Za-z
False
>>> str.isalpha()              # 检测字符串是否仅包含字母
False
>>> str.isdigit()              # 检测字符串是否仅包含数字
False
>>> str.islower()              # 检测字符串是否均为小写字母
True
>>> str.isspace()              # 检测字符串中所有字符是否均为空白字符
```

```
False
>>> str.istitle()           # 检测字符串中的单词是否为首字母大写
False
>>> str.isupper()           # 检测字符串是否均为大写字母
False
>>> str.join('HI')          # 连接字符串
'Hhi, python!I'
>>> str.upper()              # 将字符串全部转换为大写
'HI, PYTHON!'
>>> str.title()              # 将字符串中的单词首字母大写
'Hi, Python!'
>>> str.split()              # 以空格分割字符串
['hi,', 'python!']
>>> str.split(',')           # 以“,”分割字符串
['hi', 'python!']
>>> len(str)                  # 获取字符串长度
11
>>> str + 'hello'            # 使用“+”链接字符串
'hi, python!hello'
>>> str * 3                   # 使用“*”重复字符串，此处重复3次
'hi, python!hi, python!hi, python!'
>>> str * 2                   # 此处重复两次
'hi, python!hi, python!'
>>>
>>> str                       # 输出 str
'hi, python!'                 # 仍为原来的字符串
```

以上函数并不改变字符串本身，而是返回修改后的新的字符串。如果要修改原字符串，可以使用如下所示的方法。

```
>>> str = str.title()        # 将 str.title() 函数的返回值赋值给 str，即修改 str
>>> str
'Hi, Python!'
```

以上函数中使用比较复杂的为 `string.join()` 函数及 `string.split()` 函数。`string.join()` 函数将原字符串插入参数字符串中的每两个字符之间。如果参数字符串中只有一个字符，那么返回参数字符串。同样，`string.join()` 并不改变原字符串，只是返回一个新的字符串。如下所示。

```
>>> str = 'how'              # 原始字符串
>>> str.join('---')          # 将原始字符串插入“---”之中
'-how-how-'
>>> str.join('a')             # 参数字符串只有一个字符
'a'                           # 返回参数字符串
>>> str                       # 经过操作后，原字符串并没改变
'how'
```

`string.split()` 函数将字符串以指定的字符分割，如果不指定字符，则默认以空格分割字符串。其函数原型如下所示。

```
split([sep [,maxsplit]])
```


其参数含义。

- sep: 可选参数, 指定分割的字符。
- maxsplit: 可选参数, 分割次数。

```
>>> str = 'Python is wonderful!' # 定义原始字符串
>>> str.split()                  # 以空格分割字符串
['Python', 'is', 'wonderful!']  # 返回除去空格的字符串列表
>>> str.split(None, 1)           # 以空格分割, 但只分割一次
['Python', 'is wonderful!']
>>> str.split(None, 0)           # 相当于不分割
['Python is wonderful!']
>>> str.split('o',)              # 以字母“o”分割字符串
['Pyth', 'n is w', 'nderful!']
```

3.2.3 索引和分片

Python 中的字符串相当于一个不可变序列的列表。一旦声明一个字符串, 则该字符串中的每个字符都有了自己固定的位置。在 Python 中可以使用“[]”来访问字符串中指定位置上的字符, 这种方式类似于 C 语言中的数组。与数组类似, 在 Python 中, 字符串中字符的序号从 0 开始, 即 string[0] 表示字符串 string 中的第一个字符。

与 C 语言不同的是, Python 还允许以负数表示字符的序号, 负数表示从字符串尾部开始计算, 此时最后一个字符的序号为 -1, 而不是 -0。以下实例演示了如何使用“[]”访问字符串中的字符。

```
>>> str = 'abcdefg'              # 定义原始字符
>>> str[2]                       # 取字符串中序号为 2 的字符, 也就是第 3 个字符
'c'
>>> str[-2]                      # 从字符串尾部开始计算, 最后一个字符的序号为 -1,
                                # str[-2] 即取倒数第 2 个字符

'f'
>>> str[-0]                      # -0 即 0, 就是取字符串中第一个字符
'a'
>>> str[-1]                      # 取字符串中最后一个字符
'g'
>>> str[1:4]                     # 取从字符串中第 2 个字符直到第 5 个字符的内容, 但不包含
                                # 第 5 个字符

'bcd'
>>> str[1:1]                     # 由于不包含第 2 个字符, 故为空
''
>>> str[2:4]                     # 从第 3 个字符开始到第 5 个字符, 但不包含第 5 个字符
'cd'
>>> str[1:-1]                   # 从第 2 个字符开始到最后一个字符, 但不包含最后一个
'bcdef'
>>> str[0:-2]                   # 从第一个字符开始到倒数第 2 个字符, 但不包含倒数第 2 个
'abcde'
>>> str[:-2]                    # 与 str[0:-2] 相同
'abcde'
```

3.2.4 格式化字符串

由于字符串中的字符顺序是不可变的，但是在某些情况下，又要根据不同的需要修改字符串的内容。在 Python 中，可以在字符串中使用以“%”开头的字符，以在脚本中改变字符串的内容。常用的格式化字符有以下几个。

- %c: 单个字符。
- %d: 十进制整数。
- %o: 八进制整数。
- %s: 字符串。
- %x: 十六进制整数，其中的字母小写。
- %X: 十六进制整数，其中的字母大写。

以下实例演示了在字符串中使用格式化字符的方法。

```
>>> s = 'So %s day!'          # 定义字符串，在字符串中使用%s
>>> print s % 'beautiful'     # 使用 beautiful 替换%s
So beautiful day!
>>> s % 'beautiful'          # 与 print s % 'beautiful'功能相同
'So beautiful day!'
>>> '1 %c 1 %c %d' % ('+', '=', 2) # 使用多个格式化字符
'1 + 1 = 2'
>>> 'x = %x' % 0xA            # 使用%x 格式化十六进制数字，其中的字母小写
'x = a'
>>> 'x = %X' % 0xa            # 使用%x 格式化十六进制数字，其中的字母大写
'x = A'
```

3.2.5 字符串与数字相互转换

在某些情况下，字符串可能完全由数字组成，而该字符串又需要在脚本中进行算术运算，此时就可以使用 string 模块的函数 string.atoi()将字符串转换为整数。其函数的原型如下。

```
string.atoi( s[, base])
```

其参数的含义。

- s: 进行转换的字符串。
- base: 可选参数，表示将字符转换成的进制类型。

将数字转换为字符串可以使用 str()函数。以下实例演示了在 Python 中字符串与数字的相互转换。

```
>>> '10' + 4                  # 两种不同类型对象相加引发异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> string.atoi('10') + 4     # 将字符串转换为数字
14
```



```
>>> >>> '10' + str( 4 )           # 将数字转换为字符串
'104'
>>> string.atoi('13', 16)          # 将字符串转换为十六进制数
19
```

3.2.6 原始字符串（Raw String）

原始字符串是 Python 中一类比较特殊的字符串，以大写字母 R 或者小写字母 r 开始。在原始字符串中，“\”不再表示转义字符的含义。原始字符串是为正则表达式设计的，但是可以用其来方便地表示 Windows 系统下的路径。但是，如果路径以“\”结尾，那么会出错，如下所示。

```
>>> import os
>>> path = r'e:\book'               # 使用原始字符串
>>> os.listdir(path)                # 列出目录中的内容
['res', 'code', 'bak', 'temp']
>>> os.listdir('e:\book')           # 此处会出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
WindowsError: [Error 22] : 'e:\x08ook/*.*'
>>> path = R'e:\book\'              # 原始字符串中不能以“\”结尾
Traceback ( File "<interactive input>", line 1
  path = R'e:\book\'
      ^
SyntaxError: EOL while scanning single-quoted string
```

3.3 Python 数据类型——列表和元组

列表是以方括号“[]”包围的数据集合，不同成员间以“,”分隔。列表中可以包含任何数据类型，也可以包括另一个列表。列表也可以通过序号来访问其中的成员。在脚本中对列表进行排序、添加、删除等操作，改变列表中的某一成员的值。元组的特性与列表基本相同，元组是以圆括号“()”包围的数据集合。与列表不同的是，元组中的数据一旦确立就不能被改变。元组可以使用在不希望数据被其他操作改变的场合。以下实例演示了在 Python 中列表和元组的使用。Python 提供了对列表操作的强大支持，常用的操作如表 3-5 所示。

表 3-5 常用列表操作

列表操作	描述	列表操作	描述
list.append()	追加成员	list.pop()	删除列表中的成员
list.count(x)	计算列表中的参数 x 出现的次数	list.remove()	删除列表中的成员
list.extend(L)	向列表中追加另一个列表 L	list.reverse()	将列表中成员的顺序颠倒
list.index(x)	获得参数 x 在列表中的位置	list.sort()	将列表中成员排序
list.insert()	向列表中插入数据		

除此以外，在 Python 中也可以使用类似于字符串的分片和索引操作列表。而对于元组，没有上述的操作，只能对其使用分片和索引操作。以下实例演示了 Python 中对列表和元组的基本操作。

```
>>> list = [] # 定义一个空列表
>>> list.append( 1 ) # 向列表中添加成员
>>> list.count( 2 ) # 计算 2 在列表中出现的次数
0
>>> list.extend( [ 2, 3 , 5, 4] ) # 向列表中添加一个列表
>>> list
[1, 2, 3, 5, 4] # 列表值被改变
>>> list.index( 5 ) # 获得 5 在列表中的位置
3 # 从 0 开始，即第 4 个
>>> list.insert( 2, 6) # 从 0 开始，也就是在第 3 个成员处插入 6，其他成员顺次后移
>>> list
[1, 2, 6, 3, 5, 4]
>>> list.pop(2) # 删去列表中第 3 个成员
6
>>> list
[1, 2, 3, 5, 4]
>>> list.remove(5) # 删除列表中的 5
>>> list
[1, 2, 3, 4]
>>> list.reverse() # 颠倒列表的顺序
>>> list
[4, 3, 2, 1]
>>> list.sort() # 将列表中成员重新排序
>>> list
[1, 2, 3, 4]
>>> tuple = ( 'a', 'b', 'c') # 定义一个元组
>>> list.insert(4,tuple) # 向列表中插入一个元组
>>> list
[1, 2, 3, 4, ('a', 'b', 'c')]
>>> list[4] # 使用索引访问列表中第 5 个成员
('a', 'b', 'c')
>>> list[1:4] # 使用分片获得列表中第 2 个至第 5 个成员，但不含第 5 个成员
[2, 3, 4]
>>> tuple[2] # 获得元组中第 3 个成员
'c'
>>> tuple[1:-1] # 获得元组中第 2 个程序至最后一个程序，但不包含最后一个
('b',)
```

3.4 Python 数据类型——字典

字典是 Python 中比较特别的一类数据类型，以大括号“{}”包围的数据集合。字典与列表的最大不同在于字典是无序的，在字典中是通过键来访问成员的。字典也是可变的，可以包含任何其他类型，字典中的成员位置只是象征性的，并不能通过其位置来访问该成员。字

典中的成员是以“键：值”的形式来声明的。常用的字典操作如表 3-6 所示。

表 3-6 常用字典操作

字典操作	描述	字典操作	描述
dic.clear()	清空字典	dic.keys()	获得键的列表
dic.copy()	复制字典	dic.pop(k)	删除键 k
dic.get(k)	获得键 k 的值	dic.update()	更新成员
dic.has_key(k)	是否包键 k	dic.values()	获得值的列表
dic.items()	获得由键和值组成的列表		

以下实例演示了 Python 中字典的基本操作。

```
>>> dic = { 'apple':2, 'orange':1 }           # 定义一个字典
>>> dic.copy()                                # 复制字典
{'orange': 1, 'apple': 2}
>>> dic['banana'] = 5                          # 增加一项
>>> dic.items()
[('orange', 1), ('apple', 2), ('banana', 5)]   # 获得字典中成员的列表
>>> dic.pop('apple')                           # 删除“apple”，并返回其值
2
>>> dic
{'orange': 1, 'banana': 5}
>>> dic.pop('apple',3)                         # 删除“apple”，如果没有“apple”，则返回 3
3
>>> dic.keys()                                # 获得键的列表
['orange', 'banana']
>>> dic.values()                              # 获得值的列表
[1, 5]
>>> dic.update({'banana':3})                  # 更新“banana”的值
>>> dic
{'orange': 1, 'banana': 3}
>>> dic.update({'apple':2})                   # 更新“apple”的值，如果没有，则添加
>>> dic
{'orange': 1, 'apple': 2, 'banana': 3}
>>> dic['orange']                             # 通过键获取值
1
>>> dic.clear()                              # 清空字典
>>> dic
{}
```

3.5 Python 数据类型——文件

文件也可以看作是 Python 中的数据类型。当使用 Python 的内置函数 open 打开一个文件时，返回一个文件对象。其原型如下所示。

```
open(filename, mode, bufsize)
```

其参数含义如下。

- filename：要打开的文件名。
- mode：可选参数，文件打开模式。
- bufsize：可选参数，缓冲区大小。

其中 mode 可以是“r”表示以读方式打开文件，“w”表示以写方式打开文件，“b”表示以二进制方式打开文件。常用的文件操作如表 3-7 所示。

表 3-7 常用文件操作

文件操作	描述	文件操作	描述
file.read()	将整个文件读入字符串中	file.write()	向文件中写入字符串
file.readline()	读入文件的一行字符串中	file.wirtelines()	向文件中写入一个列表
file.readlines()	将整个文件按行读入列表中	file.close()	关闭打开的文件

以下实例演示了 Python 中文件的基本操作。

```
>>> file = open('c:/python.txt','w') # 打开 C 盘下的 python.txt 文件, 如果没有则创建
>>> file.write('python\n')           # 向文件中写入字符
>>> a = []                           # 定义空列表
>>> for i in range(10):               # 循环向列表中添加字符
...     s = str(i) + '\n'
...     a.append(s)
...
>>> file.writelines(a)                # 将列表写入文件
>>> file.close()                     # 关闭文件
>>> file = open('c:/python.txt','r')  # 重新以读方式打开文件
>>> s = file.read()                  # 读取整个文件
>>> print s                           # 输出文件内容
python
0
1
2
3
4
5
6
7
8
9
# 关闭文件, 为了使用 readlines 读取文件。
# 如果不关闭文件, 读取的内容为空。
# 因为文件内容已经被读入到变量 s 中。
>>> file.close()
>>> file = open('c:/python.txt','r')
>>> l = file.readlines()             # 将文件读取到列表中
```



```
>>> print l                                # 输出列表
['python\n', '0\n', '1\n', '2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n']
```

3.6 Python 基本语句

语句是 Python 脚本的基础。Python 脚本中的控制语句控制着脚本的执行流程，根据一定的条件来执行脚本中不同的语句，以完成不同的功能。

3.6.1 if 语句

if 语句是基本的条件测试语句，用来判断可能遇到的不同情况，并针对不同的情况进行操作。if 语句的基本形式如下。其在脚本中的执行过程如图 3-1 所示。

```
if <条件>:                                # 当条件为真时，执行缩进的语句，当条件为假判断 elif 的条件
    <语句>                                # 要用缩进来表示语句处于 if 语句之中
elif <条件>:                               # 当条件为真时，执行缩进的语句，当条件为假时执行 else
    <语句>
else:                                       # 前边所有的条件都为假，则执行下面的缩进语句
    <语句>
```

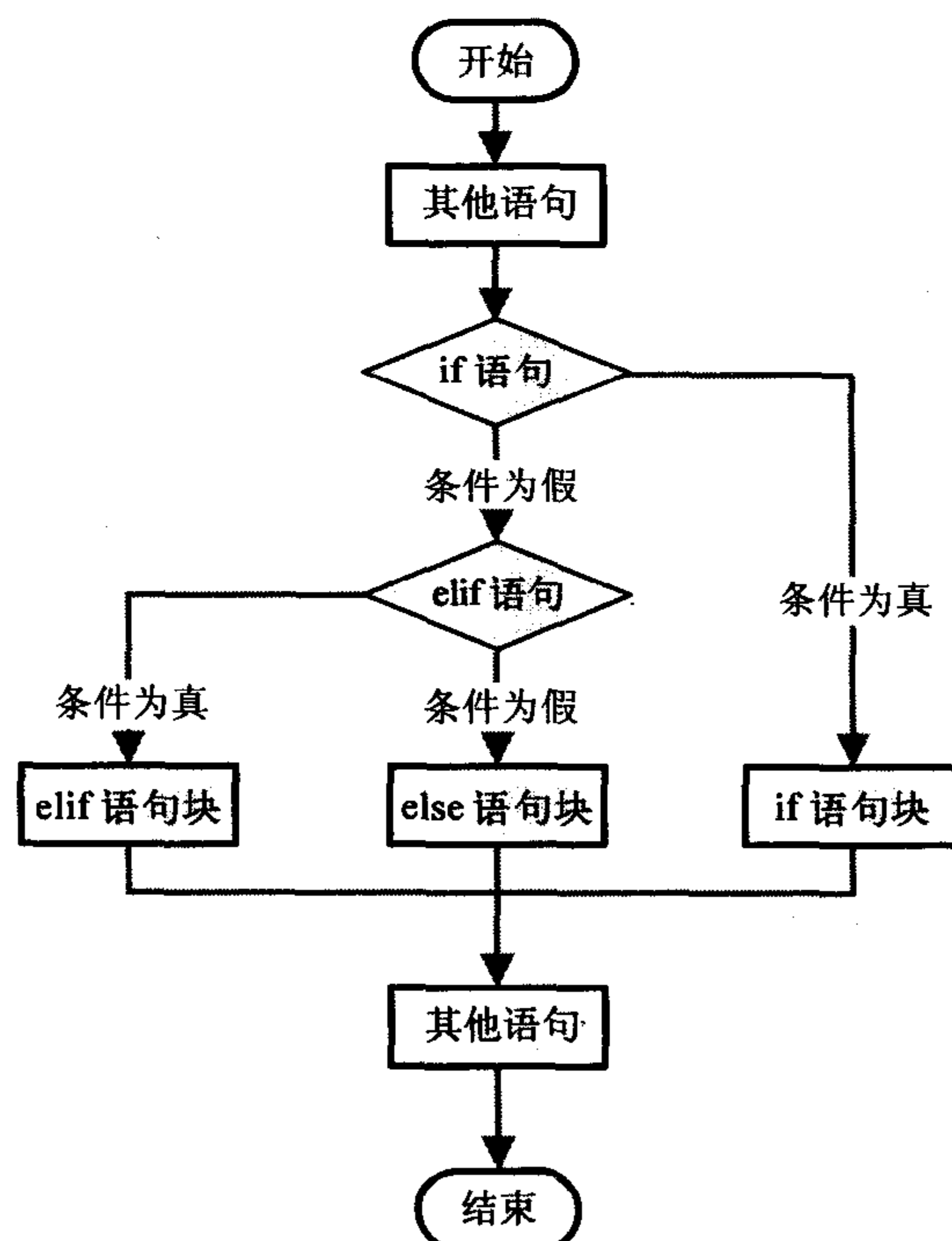


图 3-1 if 语句执行流程

在条件语句中主要使用如表 3-8 所示的几种比较运算符。

表 3-8 比较运算符

比较运算符	含 义
a == b	a 与 b 是否相等，是则返回真，否则返回假
a != b	a 是否不等于 b，是则返回真，否则返回假
a > b	a 是否大于 b，是则返回真，否则返回假
a >= b	a 是否大于等于 b，是则返回真，否则返回假
a < b	a 是否小于 b，是则返回真，否则返回假
a <= b	a 是否小于等于 b，是则返回真，否则返回假

以上几种比较运算符可以用于数字、字符串、列表、元组以及字典等。除了上述的比较运算符以外，在条件中也可以使用逻辑运算，以及一些其他的语句。以下实例演示了 Python 中基本的 if 条件测试语句。

```
>>> a = 1
>>> b = 2
>>> if a == b:                # 判断 a 与 b 是否相等
...     print 'true'          # 相等则打印 true
... else:
...     print 'false'         # 否则打印 false
...                             # 此处要多按一下回车键，脚本才会执行 if 语句
false
>>> if a < b:                 # 判断 a 是否小于 b
...     print 'true'          # 如果 a 小于 b 则打印 true
... else:
...     print 'false'         # 否则打印 fasle
...
true
>>> m = 'hi'
>>> n = 'hello'
>>> if m == n:                # 此处为字符串比较
...     print 'true'
... elif m > n:
...     print 'false'
... else:
...     print m,n
...
false
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> if l1 == l2:              # 此处为列表比较，只包含 if 语句块
...     print 'true'
...
>>> if l1 != l2:
...     print 'fasle'
...
fasle
```



```
>>> if 11 <= 12:
...     print 'true'
...
true
>>> if not 1:                                # 逻辑运算非，相当于 if 0:，即条件为假
...     print 'true'
... else: print 'false'
...
false
```

在 if 语句中还可以嵌套其他的 if 语句，被包含的 if 语句要用缩进来表示自己所包含的语句。这是 Python 独特的语法，而不像其他语言使用一对大括号“{}”来表示一个语句块，这样可以使脚本看起来更清晰，但在编写的过程中容易忽略缩进，而导致程序语法错误，或者导致结果出错。在编写 Python 脚本时最好使用具有自动缩进功能的编辑器，以保证程序正确缩进，减少敲击键盘的次数。在 if 语句中嵌套其他 if 语句的结构如下所示。

```
if <条件>:
    if <条件>:                                # 嵌入的 if 语句
        <语句>                                # 此处相对于 if 再缩进
    else:
        <语句>                                # 此处相对于 else 再缩进
elif <条件>:
    if <条件>:
        <语句>
    <其他语句>                                # 此语句未缩进，表示不属于嵌入的 if
else:
    <语句>
```

3.6.2 for 语句

for 语句是 Python 中的循环控制语句。for 语句可以用于循环遍历某一对象，它还具有一个附带的 else 块。附带的 else 块是可选的，主要用于处理 for 语句中包含的 break 语句。如果 for 循环未被 break 终止，则会执行 else 块中语句。for 语句中的 break 语句，可以在需要的时候终止 for 循环。在 for 语句中还可以使用 continue 语句。continue 语句可以跳过位于其后的语句，开始下一轮循环。for 语句的格式如下。

```
for <> in <对象集合>:
    if <条件>:
        break                                # 终止循环
    if <条件>:
        continue                            # 使用 continue 跳过其他语句，继续循环
    <其他语句>
else:
    <>                                        # 如果 for 循环未被 break 终止则执行 else 块中的语句
```

以下是一个完整的 for 循环语句。

```
>>> for i in [ 1,2,3,4,5 ]:
...     if i == 6:
...         break
```

```
...     if i == 2:
...         continue
...     print i
... else:
...     print 'all'
...
1
3
4
5
all
>>>
```

for 语句中的对象集合可以是列表、字典以及元组等。也可以通过 range()函数产生一个整数列表，以完成计数循环。range()函数的原型如下所示。

```
range([start,] stop[, step])
```

其参数含义如下。

- start 可选参数，起始数。
- stop 终止数，如果 range 只有一个参数 x，那么 range 产生一个从 0 至 x-1 的整数列表。
- step 可选参数，步长。

以下实例使用 for 和 range 函数输出 1 到 5。

```
>>> for i in range(1, 5 + 1):      # 用 range 函数产生一个包含 1 到 5 的整数列表
...     print i
...
1
2
3
4
5
```

以下实例使用 for 语句遍历一个字典。由于 for 语句遍历的是字典的键，这样就可以使用 dic[key]的形式同样遍历字典中的值。

```
>>> people = {'Tom':170, 'Jack':175, 'Kite':160, 'White':180}
>>> for name in people:
...     print people[name]
...
180
160
175
170
```

在 for 循环中，除了循环的对象可以是元组以外，循环的目标也可以是元组，可以在循环的过程中对元组进行赋值等操作。以下实例在 for 循环中使用元组。

```
>>> tt = ( ('a','b'), ('c','d'), ('e','f'), ('g','h') )
>>> for t1 in tt:      # 此处的 t1 相当于一个元组
```



```

...     print t1
...
('a', 'b')
('c', 'd')
('e', 'f')
('g', 'h')
>>> for (x,y) in tt:                # 循环的目标为一个元组
...     print x,y
...
a b
c d
e f
g h

```

以下是一个比较复杂的 for 循环，通过在 for 循环中嵌套 for 循环语句，以及 if 语句实现求解 50 至 100 之间的全部素数。

```

>>> import math                    # 导入 math 模块, 以使用求平方根的函数
>>> for i in range(50,100 + 1):    # 遍历 50 到 100
...     for t in range( 2, int(math.sqrt(i)) + 1 ): # 从 2 到 i 的平方根, 此处使用 int 将
...                                                    i 的平方根转为整数
...         if i % t == 0:          # 判断 i 能否被 2 到 i 的平方根内的数整除
...             break              # 终止循环, 即 i 不是素数
...     else:
...         print I                # 如果循环没有被 break 终止, 即 i 为
...                                素数, 打印 i
...
53
59
61
67
71
73
79
83
89
97

```

3.6.3 while 语句

while 语句是和 for 语句一样的循环控制语句。与 for 循环不同的是，while 语句中只有在测试条件为假时才会停止。在 while 的语句块中一定要包含改变条件的语句，以保证循环能够结束，避免死循环的出现。

while 语句包含与 if 语句相同的条件测试语句，如果条件为假，则终止循环。while 语句也有一个可选的 else 语句块。与 for 循环中的 else 语句块一样，当 while 循环不是由 break 语句终止的话，则会执行 else 语句块中的语句。continue 语句也可以用于 while 循环中，其作用同 if 语句中的 continue 相同，都是跳过 continue 后的语句，进入下一个循环。while 的一

般形式如下所示。

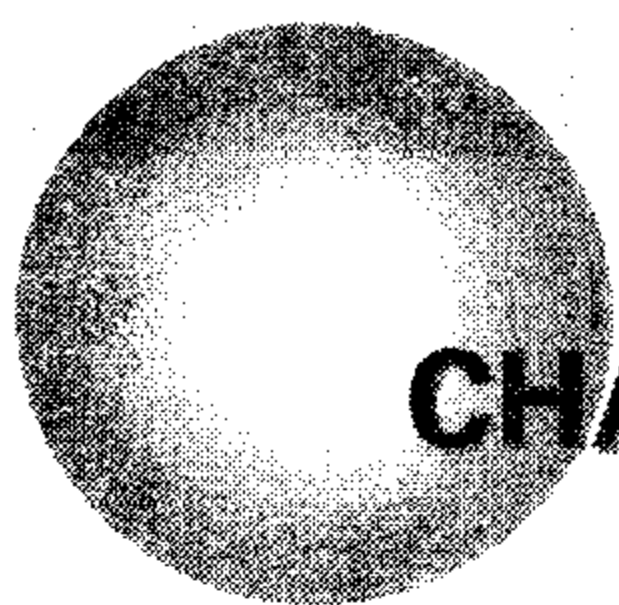
```
while <条件>:
    if <条件>:
        break           # 终止循环
    if <条件>:
        continue        # 跳过后面的语句
    <其他语句>
else:
    <语句>                # 如果循环未被 break 终止，则执行
```

while 的使用比较简单。由于其不像 for 循环可以遍历某一对象的集合，while 循环最容易出现的问题就是条件永远为真，导致死循环，因此在使用 while 循环时应仔细检查 while 语句的条件测试，避免出现死循环。以下实例使用 while 语句打印数字 1~5。

```
>>> x = 1
>>> while x <= 5:           # 条件测试
...     print x
...     x = x + 1           # 改变条件，使 x 增加，以结束循环
...
1
2
3
4
5
```

以下实例使用 while 语句访问一个列表，可以看出使用 while 语句遍历列表要比使用 for 语句遍历列表复杂一些。

```
>>> l = [ 'a', 'b', 'c', 'd', 'e' ]
>>> i = len( l )           # 获得列表长度
>>> while i != 0:          # 条件测试
...     print l[-i]        # 因为 i 从大到小递减，故使用负值，从头输出列表的值
...     i = i - 1          # 条件控制语句，避免死循环
...
a
b
c
d
e
```



CHAPTER 4

第 4 章 函数与模块

函数是一组语句的集合，用以实现某一特定的功能。函数可以简化脚本，Python 本身已提供了许多内置函数，极大地方便了脚本编写。而模块，则可以看作是一组函数的集合。很多函数库在 Python 中都是以模块的形式提供的。

4.1 函数

在编写脚本的过程中，经常要完成许多重复的工作，此时就可以将完成重复工作的语句提取出来，将其编写为函数。在脚本中可以方便地调用函数来完成这些重复的工作，而不必重复地粘贴、复制代码。在前边的章节中已经讲解了 Python 中的部分函数。在 Python 中，函数必须先声明，然后才能在脚本中使用。使用函数时，只要按照函数定义的形式，向函数传递必需的参数，就可以完成函数所实现的功能。

4.1.1 函数声明

在 Python 中，使用 `def` 可以声明一个函数。完整的函数是由函数名、参数以及函数实现语句组成的。同上一章节中所讲解的 Python 基本语句一样，在函数中也要使用缩进以表示语句属于函数体。如果函数有返回值，那么需要在函数中使用 `return` 语句，返回需要的值。声明函数的一般形式如下所示。

```
def <函数名> (参数列表):  
    <函数语句>  
    return <返回值>
```

其中参数和返回值不是必需的。很多函数可能既不需要传递参数，也没有返回值。比如如下定义的一个简单函数。

```
# hi 是所声明函数的函数名，以便在脚本中使用该函数  
# 虽然不需要传递参数，但在函数声明的时候依然要在函数名后跟一对圆括号  
>>> def hi ():  
...     print 'hi,python!'           # 缩进的语句，表示是函数内的语句  
...                                     # 函数没有使用 return 定义返回值
```

以下是一个完整的函数，实现了求一个列表中所有整数之和。其参数 `L` 为所要求解

的列表，result 就是列表中所有整数的和，最后函数使用 return 语句将 result 返回。函数如下所示。

```
>>> def ListSum( L ):
...     result = 0
...     for i in L:
...         result = result + i
...     return result
```

Python 的函数比较灵活。与 C 语言中函数的声明相比，在 Python 中声明一个函数不需要声明函数类型，也不需要声明参数的类型。在 Python 的实际处理函数的过程中也非常地灵活，不必为不同类型的参数声明多个函数，在处理不同类型数据时调用相应的函数。如下所示的函数，其功能是打印参数对象中的所有成员。

```
>>> def PrintAll ( X ):          # 声明函数
...     for x in X:              # 遍历参数
...         print x
...
>>> l = [ 1, 2, 3, 5]           # 定义一个列表
>>> PrintAll(l)                 # 打印列表中的内容
1
2
3
5
>>> t = ( 'a','b','c')          # 定义一个元组
>>> PrintAll(t)                 # 打印元组中的内容
a
b
c
```

此处只声明了一个 PrintAll(X)的函数，没有指定参数的类型。函数调用的时候，不仅可以向其传递一个列表，也可以向其传递一个元组。可以看到，不管参数为一个列表，还是一个元组，函数都被正确地执行。

这并不表示可以向函数传递任何参数，这主要还是取决于函数的实现。在 PrintAll 函数中，只使用了 for 循环语句，以及 print 函数，它们都对所操作的对象没有特别的要求，因此函数才得以正确地执行。虽然 Python 中的函数灵活性很强，但这也意味着一旦出现问题只有在脚本运行的时候才能被发现。

4.1.2 函数调用

在以 PrintAll 函数为例时，已经演示了如何调用函数。在 Python 中只要使用函数名，然后在函数名后使用圆括号将函数需要的参数包围，不同的参数以“,” 隔开。即使函数不需要参数，也要在函数名后使用圆括号。函数调用必须在函数声明之后。代码如下所示。

```
>>> def hi ():                  # 上一小节中定义的函数，需要传递参数，也没有返回值
...     print 'hi,python!'
...
```

```

>>> hi()
hi,python!
>>> hi
<function hi at 0x01124770>
>>> def ListSum( L ):
...     result = 0
...     for i in L:
...         result = result + i
...     return result
>>> l = [ 1, 2, 3, 4, 5]
>>> r = ListSum( l )
>>> print r
15
>>> r = hi()
hi,python!
>>> print r
None

```

调用函数，使用一对空括号
函数运行的结果，而不是返回值
只输入函数名，而不加括号
返回的是函数在内存中地址
上一小节中定义的函数，求解列表中的整数之和
定义一个列表
调用 ListSum 函数，传递 l 为参数，将 r 赋值为函数的返回值
输出 r 的值
将 r 赋值为 hi 函数的返回值
输出 r 的值
表示函数无返回值，也可以理解为函数返回 None

4.2 函数中的参数

在 Python 中，函数的参数可以有多种形式。例如某些函数，在其调用时需要向其传递参数，但在使用时，可以不向其传递参数，依然可以正确调用函数。在 Python 中，还可以声明一个具有任意个参数的函数。

4.2.1 参数默认值

在 Python 中可以在声明函数时，预先为参数设置一个默认值。当调用参数具有默认值的函数时，可以不向函数传递参数，而使用声明函数时设置的默认值。声明一个参数具有默认值的函数形式如下。

```

def <函数名> (参数=默认值):
    <语句>

```

以下实例定义了一个函数计算参数的立方，其参数的默认值为 5。

```

>>> def Cube ( x = 5 ):
...     return x ** 3
...
>>> Cube(2)
8
>>> Cube()
125

```

声明函数，将参数默认值设置为 5
调用函数，计算 2 的立方
调用函数，计算默认参数的立方

如果一个函数具有多个参数，而且这些参数都具有默认值，在调用函数的时候，可能仅想向最后一个参数传递值，代码如下所示。

```

>>> def Cube( x = 1, y = 2, z = 3 ):
...     return ( x + y - z ) ** 3
...
>>> Cube ( 0 )

```

声明函数
向参数传递一个值，是传递给 x

```
-1
>>> Cube ( 3, 3)                                # 向参数传递两个值，是传递给 x, y
27
>>> Cube ( , , 5 )                                # 这样是错误的
Traceback ( File "<interactive input>", line 1
      Cube ( , , 5 )
            ^
SyntaxError: invalid syntax
```

从上述实例可以看出，在 Python 中传递参数是按照声明函数时参数的顺序依次传递的。如果在调用函数时仅使用“,”表示向函数的最后一个参数传递值，会引发错误。如果需要向指定的参数传递值可以使用如下方式重新定义一下函数。

```
>>> def Cube( x = None, y = None, z= None ):      # 声明函数，将参数默认值均设为 None
...     if x == None:                             # 判断参数 x 的值是否为 None，即未传递值
...         x = 1                                 # 将 x 赋值为 1，相当于参数 x 的默认值为 1
...     if y == None:                             # 判断参数 y 的值是否为 None，即未传递值
...         y = 2                                 # 将 y 赋值为 2，相当于参数 y 的默认值为 2
...     if z == None:                             # 判断参数 z 的值是否为 None，即未传递值
...         z = 3                                 # 将 z 赋值为 3，相当于参数 x 的默认值为 3
...     return ( x + y - z ) ** 3
...
>>> Cube ( )                                     # 调用函数，使用参数的默认值
0
>>> Cube( None, None, 5)                         # 调用函数，仅 x, y 使用默认值
-8
```

除了上述方法以外，还可以使用下一小节中的方法，向指定的参数传递值。

4.2.2 参数传递

在 Python 中参数值的传递是按照声明函数时参数的顺序进行传递的。而实际上 Python 还提供了另外一种传递参数的方法——按照参数名传递值的方法。以参数名传递参数时类似于设置参数的默认值。使用按参数名传递参数的方式调用函数时，要在函数名后的圆括号里为函数的所有参数赋值。赋值的顺序不必按照函数声明时的参数顺序。代码如下例所示。

```
>>> def fun ( x, y, z):                          # 声明函数
...     return x + y - z
...
>>> fun ( 1, 2, 3)                               # 调用函数，按顺序传递参数
0
>>> fun ( z = 1, x = 2, y = 3)                   # 调用函数，按照参数名传递参数
4
```

在 Python 中，调用函数可以同时使用按顺序传递参数，以及按参数名传递参数的方式。但是，要注意，按顺序传递的参数要位于按参数名传递的参数之前，而且不能有重复的情况。代码如下例所示。

```
>>> def mysum( x, y, z):                        # 声明函数
```



```

...     return x + y + z
...
>>> mysum ( 1, z = 3, y = 2)    # 同时使用按顺序传递参数和按参数名传递参数的方式
6
>>> mysum( z = 3,y =2, 1)        # 错误的方式!按顺序传递的参数不能位于按参数名传递的参数之后
Traceback ( File "<interactive input>", line 1
SyntaxError: non-keyword arg after keyword arg (<interactive input>, line 1)
>>> mysum ( 5, z = 6, x = 7)    # 错误的方式! 参数重复, 5 已经传递给 x, 后边又将 7 传递给 x
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: mysum() got multiple values for keyword argument 'x'

```

在具有默认参数值的函数中, 使用按参数名向函数中参数传递值的方法非常方便。例如在上一小节的例子中, 如果使用按照参数名向函数传递参数的方法, 不必在函数的声明时将参数的默认值设置为 None, 可以省去函数中的判断语句。不过, 为了让函数更具通用性, 以下实例的代码依然将参数的默认值设置为 None。代码如下所示。

```

>>> def Cube( x = None, y = None, z= None ):    # 声明函数
...     if x == None:
...         x = 1
...     if y == None:
...         y = 2
...     if z == None:
...         z = 3
...     return ( x + y - z ) ** 3
...
>>> Cube(z = 5)                                # 按参数名向函数中参数 z 传递值
-8
>>> Cube( y = 6, z = 3)                        # 按参数名向函数中参数 y, z 传递值
64

```

4.2.3 可变长参数

在 Python 中, 函数可以具有任意个参数, 而不必将所有参数定义。使用可变长参数的函数, 将其所有参数保存在一个元组里, 在函数中可以使用 for 循环来处理。声明一个可变长参数的函数只需以 “*” 开头定义一个参数即可。代码如下所示。

```

>>> def mylistappend( *list ):                # 声明一个可变长参数的函数
...     l = []
...     for i in list:                          # 循环处理参数
...         l.extend( i )                      # 将所有参数中的列表合并到一起
...     return l
...
>>> a = [ 1, 2, 3 ]                            # 定义列表
>>> b = [ 4, 5, 6 ]
>>> c = [ 7, 8, 9 ]
>>> mylistappend( a, b )                       # 调用函数, 传递两个参数
[1, 2, 3, 4, 5, 6]
>>> mylistappend( a, b, c )                   # 调用函数, 传递三个参数

```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.2.4 参数引用

在 C 语言中，可以通过在参数中使用指针以达到改变参数值的作用。在 Python 中，可以在参数中使用可变对象，如列表等，来达到改变参数值的目的。代码如下所示。

```
>>> def ChangeValue1( x ):          # 此处参数 x 应为整数
...     x = x ** 2
...
>>> def ChangeValue2( x ):          # 此处参数 x 应为列表
...     x[0] = x[0] ** 2
...
>>> a = 2                           # a 为整数，其值为 2
>>> b = [2]                         # b 为列表，其第一个成员为 2
>>> ChangeValue1(a)                 # 使用函数改变 a 的值，但不成功
>>> a
2
>>> ChangeValue2(b)                 # 使用函数改变 b 成员的值
>>> b
[4]                                # 值被成功改变
```

4.3 作用域

在 Python 脚本中，不同的函数可以具有相同的参数名。在函数里已经声明的变量名，还可以在函数以外继续使用。而在脚本运行的过程中，其值并不相互影响。代码如下所示。

```
>>> def fun1( x ):                  # 声明一个函数
...     a = [ 1 ]                   # 定义一个名为 a 的列表
...     a.append(x)
...     print a
...
>>> a = [ 2, 3, 4 ]                # 在函数里定义一个名为 a 的列表
>>> fun1(2)                         # 调用函数，输出函数中列表的值
[1, 2]
>>> a                               # 输出函数里名为 a 的列表值
[2, 3, 4]                           # 两者值并不一样
```

上述实例中两个同名的列表之所以其值不同，是因为它们处于不同的作用域里。在 Python 中，作用域可以分为内置作用域、全局作用域以及局部作用域。内置作用域是 Python 预先定义的。全局作用域是所编写的整个脚本。局部作用域是某个函数内部范围。

上述实例中函数中的列表 a 处于局部作用域中，而函数外的列表 a 处于全局作用域内。局部作用域内变量的改变并不影响全局作用域内的变量，除非通过引用的形式传递参数。如果要在函数中使用函数外的变量，可以在变量名前使用 global 关键字。代码如下所示。

```
>>> def fun( x ):                  # 声明函数
...     global a                    # 使用 global 关键字声明全局变量
...     return a + x
```

```

...
>>> a = 5                # a 为全局变量，即 fun 函数中的 a
>>> fun(3)                # 调用函数
8
>>> a = 2                # 修改 a 的值
>>> fun(3)                # 再次调用函数
5                          # 返回值改变

```

4.4 lambda 表达式

lambda 表达式是 Python 中一类比较特殊的声明函数的方式，lambda 来源于 LISP 语言。使用它可以声明一个匿名函数，所谓匿名函数是指所声明的函数没有函数名，lambda 表达式就是一个简单的函数。使用 lambda 声明的函数返回一个值，在调用函数的使用直接使用 lambda 表达式的返回值。使用 lambda 声明函数的一般形式如下。

lambda 参数列表:表达式

以下实例使用 lambda 定义了一个函数，并调用这个函数。

```

>>> fun = lambda x: x * x - x    # 使用 lambda 定义一个函数，返回函数地址
>>> fun(3)                       # 调用 lambda 定义的函数
6                                # 函数返回值
>>> fun                           # fun 实际指向 lambda 定义的函数地址
<function <lambda> at 0x0111B970>

```

lambda 适用于定义小型函数。与 def 声明的函数不同，使用 lambda 声明的函数，在函数中仅包含单一的参数表达式，而不能包含其他的语句。在 lambda 中也可以调用其他的函数。代码如下所示。

```

>>> def show():                # 使用 def 声明 show 函数
...     print 'lambda'
...
>>> f = lambda:show()          # 在 lambda 中调用 show 函数
>>> f()                        # 调用使用 lambda 生成的函数
lambda
>>> def shown( n ):            # 使用 def 声明 shown 函数
...     print 'lambda' * n
...
>>> fn = lambda x : shown( x ) # 在 lambda 中向 shown 函数传递值
>>> fn(2)                     # 调用 lambda 生成的函数
lambdalambda
>>> def userreturn( x ):        # 使用 def 声明 userreturn 函数
...     return x*2
...
>>> fr = lambda x:userreturn(x) * x # 在 lambda 函数中使用 userreturn 函数的返回值
>>> fr(3)
18
>>> fun = lambda x: print x     # 不能在 lambda 中使用 print 语句
Traceback ( File "<interactive input>", line 1

```



```
fun = lambda x: print x
                ^
SyntaxError: invalid syntax
>>> fun = lambda x: if x <= 0 : x = -x      # 不能在 lambda 表达式中使用其他语句
Traceback ( File "<interactive input>", line 1
    fun = lambda x: if x <= 0 : x = -x
                    ^
SyntaxError: invalid syntax
```

4.5 模块

Python 中的模块实际上就是包含函数或者类的 Python 脚本。对于一个大型的脚本经常需要将功能细化，然后在不同的脚本中实现。在其他的脚本中以模块的形式使用细化的功能，这样便于脚本的维护。

4.5.1 模块概述

模块是包含函数和其他语句的 Python 脚本文件，它以“.py”为后缀名，也就是 Python 脚本的后缀名。用作模块的 Python 脚本与其他的脚本并没有什么区别。在 Python 中可以通过导入模块，然后使用模块中提供的函数或者数据。

1. 导入模块

在 Python 中可以使用以下两种方法导入模块或者模块中的函数。

- import: 模块名。
- import: 模块名 as 新名字。
- from: 模块名 import 函数名。

其中使用 import 是将整个模块导入，而使用 from 则是将模块中某一个函数或者名字导入，而不是整个模块。使用 import 和 from 导入模块还有一个不同之处：使用 import 导入模块时，要使用模块中的函数，则必须以模块名加“.”然后是函数名的形式调用函数；而使用 from 导入模块时，则可以直接使用模块中的函数名调用函数。

以下实例分别使用 import 和 from 导入模块。

```
>>> import string                                     # 使用 import 导入 string 模块
>>> string.capitalize('use modules ')                # 使用 string 模块中的 capitalize 函数
'Use modules '
>>> capitalize('use modules ')                       # 直接使用 capitalize 名字调用函数
Traceback (most recent call last):                   # 函数调用失败
  File "<interactive input>", line 1, in <module>
NameError: name 'capitalize' is not defined
>>> from math import sqrt                             # 使用 from 导入 math 模块中的 sqrt 函数
>>> sqrt(9)                                            # 直接使用 sqrt 名字调用函数
3.0
>>> math.sqrt(9)                                     # 错误的方法
```

```
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'math' is not defined
```

当需要使用模块中的函数时，使用 `from` 导入模块要方便得多，不用在调用函数时使用模块名。如果需要使用模块中的所有函数，则可以在 `from` 中使用 “*” 通配符，表示导入模块中的所有函数。代码如下所示。

```
>>> from string import capitalize           # 仅从 string 模块中导入 capitalize
>>> capitalize('use moudles')
'Use moudles'
>>> split('use moudles')                   # 调用 string 模块中的 split 函数出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
NameError: name 'split' is not defined
>>> from string import *                   # 重新从 string 模块中导入所有函数
>>> split('use moudles')                   # 重新调用 split 函数
['use', 'moudles']
```

在 Python 中还可以通过使用内置函数 `reload` 重新载入模块。`reload` 可以在模块被修改的情况下不必关闭 Python 而重新载入模块。在使用它重载模块时，该模块必须已经事先被导入。

2. 编写一个模块

编写一个 Python 模块是十分简单的事情。以下是一个简单的模块，在模块中只包含一个函数，这个函数只打印 “I am a module!”。将该模块保存为 `mymodule.py`。代码如下所示。

```
def show():                                # 声明 show 函数
    print 'I am a moudle!'
```

编写一个调用函数 `show` 的脚本，将其保存为 `usemodule.py`。代码如下所示。

```
import mymodule                            # 导入模块
mymodule.show()                            # 调用模块中的 show 函数
```

脚本运行后输出如下所示。

```
I am a moudle!
```

除了在模块中声明函数以外，还可以在模块中定义变量。模块中的变量同样可以在其他脚本中使用。以下代码在 `mymodule.py` 中添加了一个名为 `name` 的变量。

```
def show():
    print 'I am a moudle!'
name = 'mymodule.py'                      # 在模块中添加一个 name 变量
```

以下代码为修改的 `usemodule.py`，其中使用了 `mymodule.py` 中的 `name` 变量。

```
import mymodule
mymodule.show()
print mymodule.name                       # 打印模块中的 name 变量
mymodule.name = 'usemodule.py'           # 将 name 变量重新赋值
print mymodule.name
```

脚本运行后输出如下所示。

```
I am a moudle!
```

```
mymodule.py
usemodule.py
```

4.5.2 模块查找路径

编写好的模块只有被 Python 找到才能被导入。上一节中编写的模块以及调用模块的脚本位于同一个目录中。如果在该目录中新建一个 module 目录，并且把 mymodule.py 转移到 module 目录中，再次运行 usemodule.py，输出如下。

```
E:\book\code\第 4 章函数与模块>usemodule.py
Traceback (most recent call last):
  File "E:\book\code\第 4 章函数与模块\usemodule.py", line 1, in <module>
    import mymodule
ImportError: No module named mymodule
```

运行脚本出错，Python 解释器没有找到 mymodule 模块。在导入模块时，Python 解释器首先在当前目录中查找导入的模块。如果未找到模块，Python 解释器会从 sys 模块中的 path 变量指定的目录查找导入模块。如果在以上所有目录中均未找到导入的模块，则会出错。以下实例使用 sys.path 输出 Python 的模块查找路径。

```
>>> import sys
>>> sys.path
['', 'C:\\WINDOWS\\system32\\python25.zip',
'D:\\Python25\\DLLs', 'D:\\Python25\\lib',
'D:\\Python25\\lib\\plat-win', 'D:\\Python25\\lib\\lib-tk',
'D:\\Python25\\Lib\\site-packages\\pythonwin',
'D:\\Python25', 'D:\\Python25\\lib\\site-packages',
'D:\\Python25\\lib\\site-packages\\win32', 'D:\\Python25\\lib\\site-packages\\win32\\lib']
```

在脚本中可以向 sys.path 添加模块查找路径。如下所示脚本将当前目录下的 module 子目录添加到 sys.path 中，并从 module 目录中导入 mymodule 模块。代码如下所示。

```
import os
import sys
modulepath = os.getcwd() + '\\module'
sys.path.append(modulepath)
print sys.path
import mymodule
mymodule.show()
print mymodule.name
mymodule.name = 'usemodule.py'
print mymodule.name
```

运行脚本后输出如下。

```
['E:\\book\\code', 'C:\\WINDOWS\\system32\\python25.zip', 'D:\\Python25\\DLLs',
'D:\\Python25\\lib', 'D:\\Python25\\lib\\plat-win', 'D:\\Python25\\lib\\lib-tk',
'D:\\Python25', 'D:\\Python25\\lib\\site-packages', 'D:\\Python25\\lib\\site-packages\\win32',
'D:\\Python25\\lib\\site-packages\\win32\\lib', 'D:\\Python25\\lib\\site-packages\\Pythonwin', 'E:\\book\\code\\module']
I am a moudle!
mymodule.py
```



```
usemodule.py
```

从输出可以看出，当前路径也被添加到了 `sys.path` 路径列表中，这说明 Python 其实是按照 `sys.path` 中的路径来查找模块的。之所以首先在当前目录中查找，是因为 Python 解释器在运行脚本前将当前目录添加到 `sys.path` 路径列表中了。

4.5.3 模块编译

在上一节的例子中，运行完 `usemodule.py`，会发现 `moudle` 目录中除了 `mymodule.py` 文件以外还多了一个 `mymodule.pyc` 文件。其中 `mymodule.pyc` 就是 Python 将 `mymodule.py` 编译成字节码的文件。虽然 Python 是脚本语言，但 Python 可以将脚本编译成字节码的形式。对于模块而言，Python 总是在第一次调用后将其编译成字节码的形式，以提高脚本的启动速度。

Python 在导入模块时会查找模块的字节码文件，如果存在，则将编译后的模块的修改时间同模块的修改时间相比较。如果两者的修改时间不相符，那么 Python 将重新编译模块，以保证两者内容相符。被编译的脚本也是可以直接运行的。

没有必要去刻意编译 Python 脚本。不过，由于 Python 是脚本，如果不想将源文件发布，可以发布编译后的脚本，这样可以起到一定的保护作用。

对于不作为模块的脚本而言，Python 不会在运行脚本后将其编译成字节码的形式。如果想将其编译，可以使用 `compileall` 模块。如下所示代码可以将上一节中的 `usemodule.pyc` 编译成 “.pyc” 文件。

```
# file: compile.py
#
import py_compile;                               # 导入 py_compile 模块
py_compile.compile('usemodule.py');               # 编译 usemodule.py
```

运行 `compile.py` 后，可以看到当前目录中多了一个 `usemodule.pyc` 文件。运行 `usemodule.pyc` 后输出如下。可以看到其输出与上一节的输出一样。编译后生成的 `usemodule.pyc` 并没有改变程序功能，只是以 Python 字节码的形式存在。脚本运行后输出如下所示。

```
['E:\\book\\code', 'C:\\WINDOWS\\system32\\python25.zip', 'D:\\Python25\\DLLs',
'D:\\Python25\\lib', 'D:\\Python25\\lib\\plat-win', 'D:\\Python25\\lib\\lib-tk',
'D:\\Python25', 'D:\\Python25\\lib\\site-packages', 'D:\\Python25\\lib\\site-pa
ckages\\win32', 'D:\\Python25\\lib\\site-packages\\win32\\lib', 'D:\\Python25\\l
ib\\site-packages\\Pythonwin', 'E:\\book\\code\\module']
I am a moudle!
mymodule.py
usemodule.py
```

另外可以通过 Python 的命令行选项将脚本优化编译。Python 编译的优化选项有以下两个。

- `-O`：该选项对脚本的优化不多，编译后的脚本以 “.pyo” 为扩展名。凡是以 “.pyo” 为扩展名的 Python 字节码都是经过优化的。

- -OO: 该选项对脚本优化的程度较大。使用该标志可以使编译的 Python 脚本更小。使用该选项可以导致脚本运行错误, 因此, 应谨慎使用。

可以通过在命令行中输入以下命令将 usemodule.py 优化编译。

```
python -O compile.py
python -OO compile.py
```

4.5.4 模块独立运行——_name_属性

每个 Python 脚本在运行时都有一个 _name_ 属性 (name 前后均是两条下划线)。在脚本中通过对 _name_ 属性值的判断, 可以让脚本在作为导入模块和独立运行时都可以正确运行。在 Python 中如果脚本作为模块被导入, 则其 _name_ 属性被设置为模块名。如果脚本独立运行, 则其 _name_ 属性被设置为 “_main_”。因此可以通过 _name_ 属性来判断脚本的运行状态。

如下所示脚本 mymodule2.py 既可以自己运行, 也可以作为模块被其他脚本导入。

```
# file: mymodule2.py
#
def show():
    print 'I am a module!'
if _name_ == '_main_':
    show()
    print 'I am not a module!'
```

如下所示脚本 usemodule2.py 调用 mymodule2 模块。

```
# file: usemodule2.py
#
import mymodule2
mymodule2.show()
print 'my _name_ is', _name_
```

运行 usemodule2.py 后, 输出如下所示。

```
I am a module!
my _name_ is _main_
```

运行 mymodule2.py 后, 输出如下所示。

```
I am a module!
I am not a module!
```

4.5.5 dir()函数

如果需要获得导入模块中的所有定义的名字、函数等, 可以使用内置函数 dir() 来获得模块所定义的名字的列表。如下所示代码获得 sys 模块中的名字。

```
>>> import sys
>>> dir(sys)
['_displayhook_', '_doc_', '_excepthook_', '_name_', '_stderr_', '_stdin_',
'_stdout_', '_current_frames_', '_getframe_', 'api_version', 'appargv', 'appargvoffset',
'argv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats', 'copyright',
```

```
'displayhook', 'dllhandle', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
'exec_prefix', 'executable', 'exit', 'exitfunc', 'getcheckinterval', 'getdefaultencoding',
'getfilesystemencoding', 'getrecursionlimit', 'getrefcount', 'getwindowsversion', 'hexversion',
'last_traceback', 'last_type', 'last_value', 'maxint', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'subversion', 'version', 'version_info', 'warnoptions', 'winver']
```

dir()函数的原型如下所示。

```
dir([object])
```

其参数含义如下。

- object 可选参数，要列举的模块名。

如果不向 dir()函数传递参数，那么 dir()函数将返回当前脚本的所有名字列表，如下所示。

```
>>> a = [ 1, 3, 6 ]           # 定义一个列表
>>> b = 'python'             # 定义一个字符串
>>> dir()                     # 使用 dir()函数获得当前脚本所有名字列表
['_builtins_', '_doc_', '_name_', 'a', 'b', 'pywin']
>>> def fun():                # 定义一个函数
...     print 'Python'
...
>>> dir()                     # 再次使用 dir()函数
['_builtins_', '_doc_', '_name_', 'a', 'b', 'fun', 'pywin']
```

4.6 模块包

在 Python 中可以使用包来管理多个模块。使用 Python 中的模块包可以通过路径导入模块。使用包的好处在于可以有效地避免名字冲突，便于包的维护管理。

1. 包的组成

包可以看作是处于同一目录中的模块。在 Python 中首先使用目录名，然后再使用模块名导入所需要的模块。在包的每个目录中都必须包含一个名为“_init_.py”（init 的前后均是两条下划线）的文件。“_init_.py”可以是一个空文件，仅用于表示该目录为一个包。

“_init_.py”的主要用途是设置“_all_”变量以及包含包初始化所需的代码。对于在导入包内所有名字时在 from 中使用“*”通配符的情况，在“_init_.py”设置“_all_”变量可以保证名字的正确导入。一个简单的 Python 包的目录组成如图 4-1 所示。

在图 4-1 所示的包中，如果需要导入 B 目录中的 a.py 模块，可以使用如下语句。

```
from A.B import a           # 使用 from 导入模块
import A.B.a                # 使用 import 导入模块
```

有了包的概念就可以很好地解决模块查找路径的问题。只要将所有的模块放在当前目录中的某一文件夹内，然后在该文件夹中新建一个空的“_init_.py”文件即可，而不必像前边的例子将子目录的路径添加到 sys.path 列表中。

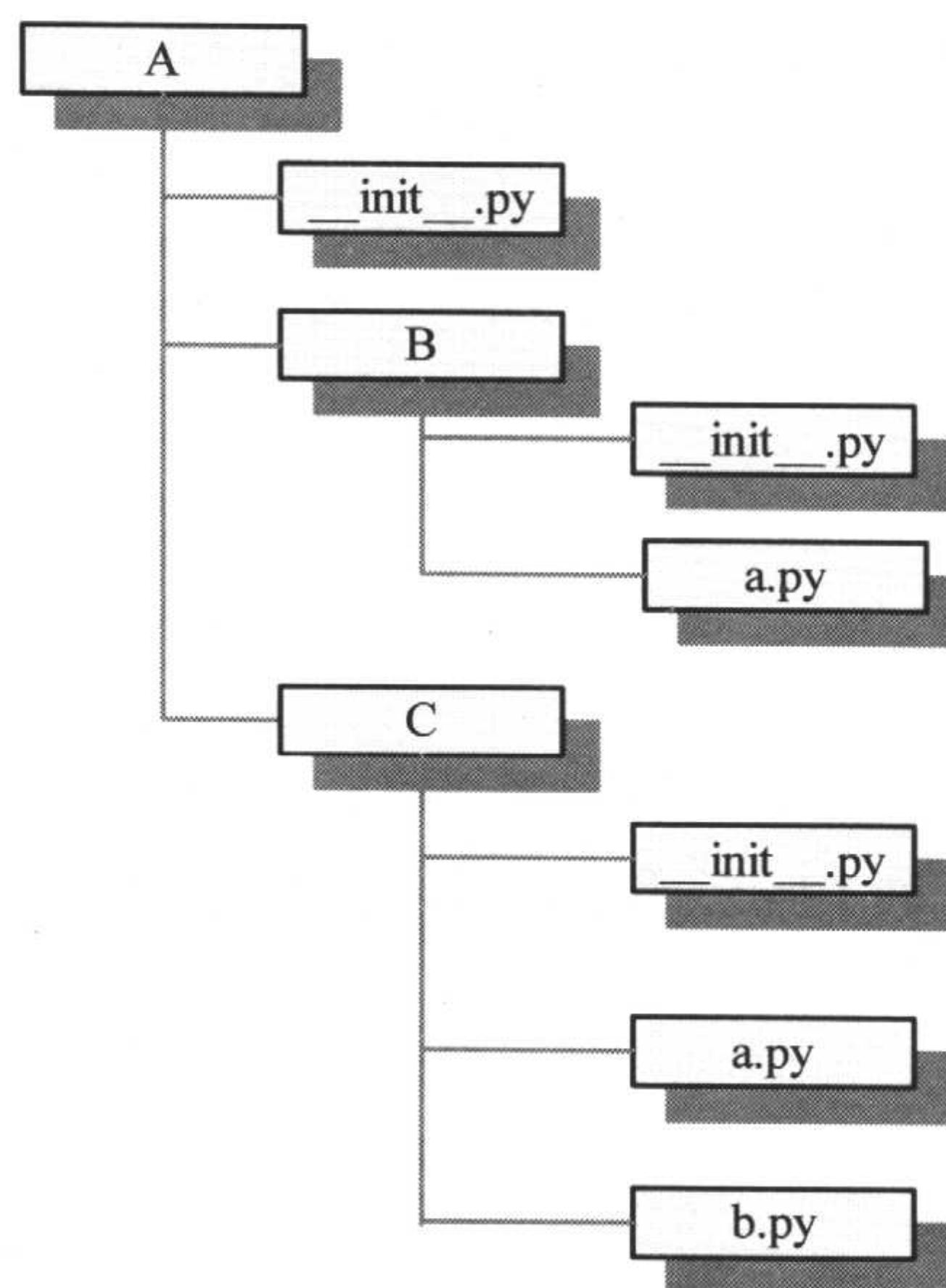


图 4-1 Python 包的组成

2. 包的内部引用

Python 包中的模块也可能需要相互引用。对于图 4-1 中所示的位于 C 目录中的 b.py，如果要引用同样位于 C 目录中的 a.py 可以使用如下语句。

```
import a
```

如果位于 C 目录中的 b.py 要引用位于 B 目录中的 a.py，则需要使用如下语句。

```
from A.B import a
```

第 5 章 正则表达式

正则表达式是用某种模式去匹配一类具有共同特征的字符串，正则表达式主要用于处理文本。它能够使文本处理变得简单起来，尤其对于复杂的查找、替换这样的工作，使用正则表达式会非常快地完成。在流行的文本编辑器里如 Emacs、Vim 等大都支持正则表达式。

5.1 正则表达式概述

在 Python 的较早版本中使用 regex 模块来完成正则表达式的操作。在新的版本中已经删除了 regex 模块。当前的 Python 主要使用 re 模块进行正则表达式的操作。re 模块提供了 Perl 风格的正则表达式。re 模块具有更好的可读性以及更强的功能。

5.1.1 基本元字符

元字符是正则表达式中含有的字符。在正则表达式中可以在字符串中使用元字符用以匹配字符串的各种可能的情况。常用的元字符如表 5-1 所示。

表 5-1

元字符表

元 字 符	含 义
.	匹配除换行符以外的任何单个字符，如“r.d”会匹配“red”、“r d”等，但不会匹配“read”
*	匹配位于*之前的任意个字符，如“r*ed”会匹配“rred”、“rrred”、“red”等
+	匹配位于+之前的一个或多个字符，如“r+ed”会匹配“rred”、“rrred”，但不会匹配“red”
	匹配位于 之前或者之后的字符，如“red blue”会匹配“red”、“blue”
^	匹配行首
\$	匹配行尾
?	匹配位于? 之前的零个或一个字符，如“r?ed”会匹配“rred”、“red”等，但不会匹配“rrred”
\	表示位于\之后的为转义字符
[]	匹配位于[]中的任何一个字符，如 r[ae]d, 会匹配“rad”、“red”
()	将位于()内的内容当作一个整体
{}	按{}中的次数进行匹配

元字符还可以配合起来使用。“.”可以匹配任意个字符，如“r.*d”会匹配“rd”、“red”、“read”等。“+”可以匹配任意的一个或者多个字符，如“r.+d”会匹配“red”、“read”，但不会匹配“rd”。“?”可以匹配任意的零个或一个字符，如“r.?d”会匹配“rd”、“red”，但不会匹配“read”。

“^”匹配行首，对于如下所示的一段文字，“^red”只会匹配文中的第 3 个“red”。而对于“red\$”则只会匹配文中的第 2 个“red”。

```
a red hat
blue and red
red and blue
```

在“[]”中还可以使用“-”来表示某一范围。例如在“[]”中，“a-z”表示从“a”到“z”的所有小写字符，同样“A-Z”表示从“A”到“Z”的所有大写字母，而“0-9”表示从“0”到“9”的数字。“[a-zA-Z0-9]”表示任意的字母或者数字。

5.1.2 常用正则表达式分析

使用正则表达式可以简化程序设计。例如，在文本文件中包含一些联系人的手机号码，如果需要将使用联通和移动的号码的人区分出来，可以使用正则表达式分别匹配。对于联通的手机号，第 3 位是 0、1、2 或者 3。而移动的则是 4~9 中的某一个数字，可以据此来判断。如下正则表达式匹配联通的手机号。

```
13[0-3][0-9]{8}
```

其中 13 表示匹配手机号的前两位。[0-3]匹配手机号的第 3 位，表示 0~3 之间的任何一个数字。[0-9]{8}组合起来匹配手机号剩下的 8 位。[0-9]表示 0~9 之间的任何一个数字，{8}表示匹配[0-9]8 次，也就是匹配任意一个 8 位数。如下正则表达式匹配移动的手机号。

```
13[4-9][0-9]{8}
```

其与匹配联通的手机号唯一不同的地方是第 3 位是用[4-9]。[4-9]表示 4~9 之间的任何一个数字。

同样如果联系人的信息中还包含邮政编码，而又需要将其按地区来区分的话也可以使用正则表达式。对于邮政编码的匹配则相对简单一些，例如北京的邮政编码前 3 位为 100。采用与匹配手机号同样的方式，只需在 100 之后匹配任意一个 3 位数，如下所示。

```
100[0-9]{3}
```

使用正则表达式匹配数字十分方便，但如果匹配字符串，则需要考虑较多的情况。例如，需要找出某一文件中所有的网址则比较复杂。以“http://www.python.org”为例，其可以分成 4 部分，首先是“http://”，然后为“www”，再就是站名“python”，剩下的是后缀“org”。可能有的网址书写完整，具有以上 4 个部分。而有些网址则不正规，不含“http://”部分。

将“http://www”当作一个部分，其可能的情况为“http://www”或者“www”，这一部分的正则表达式匹配可以写为如下所示的形式。

```
(http://www|www) # 使用“()”表示其为一个整体，使用“|”表示其中任何一个满足则匹配
```


中间的站名作为一部分，这部分可能为字母、数字或者“-”，因此该部分的正则表达式匹配可以写成如下所示的形式。

`[a-z0-9-]*` # `[a-z0-9-]`表示字母、数字或者“-”，“*”表示匹配0个或多个前边的字符剩下的后缀，考虑到可能为两个或三个的字符，因此写成如下形式。

`[a-z]{2,3}` # `{2,3}`表示匹配两次或三次，即匹配由两个或三个字母组成的字符串

由于其中还包含“.”，而在正则表达式中其具有特殊含义，需要将其使用“\”转义。完整的正则表达式如下所示。

`(http://www|www)\.[a-z0-9-]*\.[a-z]{2,3}`

此处没有考虑诸如“com.cn”这样的形式，读者可以试着自己将其完成。

5.2 re 模块函数应用

Python 中的 re 模块提供了对正则表达式的支持。虽然 Python 中有一个 string 模块用来对字符串进行处理，但 string 模块只能完成简单的操作。而使用 re 模块可以完成对复杂字符串的操作，它提供了以下几类对字符串进行操作的函数。

5.2.1 匹配和搜索

`re.match()`函数用于在字符串中匹配正则表达式，如果匹配成功，则返回 `MatchObject` 对象实例。`re.search()`函数用于在字符串中查找正则表达式，如果找到，则返回 `MatchObject` 对象实例。`re.findall()`函数用于在字符串中查找所有符合正则表达式的字符串，并返回这些字符串的列表。如果在正则表达式中使用了组，则返回一个元组。

`re.match()`函数和 `re.search()`函数的作用基本一样。不同的是，`re.match()`函数只从字符串中第一个字符开始匹配。而 `re.search()`函数则搜索整个字符串。以上 3 个函数的原型如下所示。

```
re.match( pattern, string[, flags])
re.search( pattern, string[, flags])
findall( pattern, string[, flags])
```

其参数含义如下。

- `pattern`: 匹配模式。
- `string`: 要进行匹配的字符串。
- `flags`: 可选参数，进行匹配的标志。

参数 `flags` 可以是以下选项。

- `re.I`: 忽略大小写。
- `re.L`: 根据本地设置而更改 `\w`、`\W`、`\b`、`\B`、`\s`，以及 `\S` 的匹配内容。
- `re.M`: 多行匹配模式。
- `re.S`: 使“.”元字符匹配换行符。

- re.U: 匹配 Unicode 字符。
- re.X: 忽略 pattern 中的空格, 并且可以使用 “#” 注释。

上述的几个编译标志可以同时使用。同时使用几个编译标志时, 需要使用 “|” 对并用的编译标志进行运算。以下实例使用上述函数进行匹配和搜索。

>>> import re	# 导入 re 模块
>>> s = 'Life can be good'	# 定义字符串
>>> print re.match('can',s)	# 在字符串中匹配 “can”
None	# 输出为 None 表示未找到
>>> print re.search('can',s)	# 在字符串中搜索 “can”
<_sre.SRE_Match object at 0x010DAB48>	# 返回一个 Matchobject 对象, 表示找到
>>> print re.match('l.*',s)	# 匹配任何以字母 “l” 开头的字符串
None	# 表示未找到
>>> print re.match('l.*',s,re.I)	# 此处设置忽略大小写
<_sre.SRE_Match object at 0x010DAC28>	# 返回一个 Matchobject 对象, 表示找到
>>> re.findall('[a-z]{3}',s)	# 查找所有 3 个字母的字符串
['ife', 'can', 'bad']	
>>> re.findall('[a-z]{1,3}',s)	# 查找所有由 1 到 3 个字母组成的字符串
['ife', 'can', 'be', 'bad']	

5.2.2 替换函数

re.sub()函数用于替换字符串中符合正则表达式的内容, 它返回替换后的字符串。re.subn()函数与 re.sub()函数相同, 只不过 re.subn()函数返回一个元组。其函数原型分别如下所示。

```
re.sub( pattern, repl, string[, count])
re.subn( pattern, repl, string[, count])
```

其参数含义如下。

- pattern: 正则表达式模式。
- repl: 要替换成的内容。
- string: 进行内容替换的字符串。
- count: 可选参数, 最大替换次数。

>>> import re	# 导入 re 模块
>>> s = 'Life can be bad'	# 定义字符串
>>> re.sub('bad','good',s)	# 用 “good” 替换 “bad”
'Life can be good'	
>>> re.sub('bad be','good',s)	# 用 “good” 替换 “bad” 或者 “be”
'Life can good good'	
>>> re.sub('bad be','good',s,1)	# 用 “good” 替换 “bad” 或者 “be”, 但只替换一次
'Life can good bad'	
>>> re.subn('bad be','good',s,1)	# 用 “good” 替换 “bad” 或者 “be”, 但只替换一次
('Life can good bad', 1)	# 返回由替换后的字符串和替换的数组成的元组
>>> r = re.subn('bad be','good',s)	# 用 “good” 替换 “bad” 或者 “be”
>>> print r[0]	# 输出元组第一项
Life can good good	
>>> print r[1]	# 输出元组第二项
	1

5.2.3 分割字符串函数

re.split()函数用于分割字符串，它返回分割后的字符串列表。其函数原型如下。
re.split(pattern, string[, maxsplit = 0])
其参数含义如下。

- pattern：正则表达式模式。
- string：要分割的字符串。
- maxsplit：可选参数，最大分割次数。

以下实例使用上述模块对字符串进行操作。

```
>>> import re                                # 导入 re 模块
>>> s = 'Life can be bad'                    # 定义字符串
>>> re.split(' ',s)                          # 使用空格分割字符串(注意:单引号之间有一个空格)
['Life', 'can', 'be', 'bad']
>>> r = re.split(' ',s,1)                    # 只分割一次
>>> for i in r:                               # 遍历分割后返回的列表
...     print i
...
Life
can be bad
>>> re.split('b',s)                          # 使用字母“b”分割字符串
['Life can ', 'e ', 'ad']
```

5.3 正则表达式对象

使用 re.compile()函数将正则表达式编译生成正则表达式对象实例后，可以使用正则表达式对象实例提供的属性和方法对字符串进行处理。

5.3.1 以“\”开头的元字符

除了基本的元字符以外还有一类以“\”开头的元字符。以“\”开头的元字符主要表示某一类型的集合，比如数字的集合、字母的集合等。常用的以“\”开头的元字符如表 5-2 所示。

表 5-2 以“\”开头的元字符

转义字符	含 义	转义字符	含 义
\b	匹配单词头或者单词尾	\s	匹配任何空白字符
\B	与\b 含义相反	\S	匹配任何非空白字符
\d	匹配任何数字	\w	匹配任何字母、数字，以及下划线
\D	匹配任何非数字	\W	匹配任何非字母、数字，以及下划线

“\”开头的元字符也可以与其他的元字符配合使用。例如“\d*”可以匹配任何由数字组成的字符串。其中“\d”相当于[0-9]，“\w”相当于[a-zA-Z0-9_]。以下实例演示在正则表达式中使用以“\”开头的元字符。

```
>>> import re                                # 导入 re 模块
>>> s = 'Python can run on Windows'          # 定义字符串
>>> re.findall('\bo.+?\b',s)                  # 查找首字母为“o”的单词
['on']
>>> re.findall('\Bo.+?',s)                    # 查找含字母“o”的单词，但“o”不是单词的首字母
['on', 'ow']
>>> re.findall('\so.+?',s)                    # 使用空白字符来匹配首字母为“o”的单词
[' on']
>>> re.findall('\b\w.+?\b',s)                 # 返回的字符串中含有一个空格
['Python', 'can', 'run', 'on', 'Windows']
>>> re.findall('\d\.\d','Python 2.5')         # 查找 x.x 的数字形式
['2.5']
>>> re.findall('\D+', 'Python 2.5')           # 此处匹配 2.5
['Python ', '.']
>>> re.split('\s',s)                          # 查找不含数字的字符
['Python', 'can', 'run', 'on', 'Windows']
>>> re.split('\s',s,1)                        # 使用空白字符分割字符串
['Python', 'can run on Windows']
>>> re.findall('\d\w+?', 'abc3de')            # 使用空白字符分割字符串，但只分割一次
['3d']
```

在上述的例子中，使用的是“\bo.+?\b”来匹配首字母为“o”的单词，而不是直接使用“\bo.+”或者“\bo.*”来匹配。如果使用“\bo.+”或者“\bo.*”，则输出如下。

```
>>> re.findall('\b\o.+?\b',s)
['on Windows']
>>> re.findall('\b\o.*?\b',s)
['on Windows']
```

可以看到，本来预想的是“on”后有一个空格，应该只匹配到“on”，然而最终结果连之后的“Windows”也匹配了。这是因为“+”“*”等元字符为“贪婪”模式的元字符，它们尽可能匹配更多的字符。为了避免“+”“*”等元字符过多地匹配，可以在其之后使用“非贪婪”模式的“?”，或者使用“{ }”指定匹配的次数。

5.3.2 编译正则表达式

re 模块中包含一个 re.compile() 函数，可以使用 re.compile() 函数将正则表达式编译生成一个 RegexObject 对象实例。然后通过生成的 RegexObject 对象实例对字符串进行操作，如查找，替换等。re.compile() 的函数原型如下所示。

```
compile(pattern[, flags])
```

其参数含义如下。

- pattern: 正则表达式的匹配模式。
- flags: 可选参数，编译标志。

以下实例编译生成一个 `RegexObject` 对象实例。

```
>>> import re                                # 导入 re 模块
>>> re.compile('a*b',re.I|re.X)              # 编译正则表达式，忽略大小写和模式中的空格
<_sre.SRE_Pattern object at 0x011232F0>
# 使用 re.X 编译标志表示在匹配模式中忽略注释已经空格等
>>> re.compile('''
... \b          # 匹配单词开始
... A?          # 以 A 或 AA 开头
... \d          # 匹配一个数字
... \w*         # 匹配任意字符
...            # 一个空行
... \b          # 匹配单词结束
... ''',re.X)
<_sre.SRE_Pattern object at 0x01093BD8>
```

5.3.3 使用原始字符串

原始字符串在第3章中已经提及。原始字符串是为正则表达式设计的，以提高正则表达式的可读性，减少“\”在正则表达式中的数目。

由于在正则表达式中也要使用以“\”开头的字符以表示某些特殊的含义，而在字符串中，转义字符也是以“\”开头，这就导致了冲突。例如在正则表达式中，“\b”表示匹配一个单词的开始或者结束，而在字符串中，“\b”则表示退格。如果在正则表达式中使用“\b”，则应该写成“\\b”。在 `re.compile()` 中使用“\b”的正确写法如下所示。

```
re.compile('\\ba.?')
```

如果使用原始字符串，则写法如下所示。

```
re.compile(r'\\ba.?')
```

如果要在正则表达式中匹配一个以“\”开头的字符串，比如“\word”，则首先要将“\”转义，以避免元字符“\w”，应将其写成“\\word”。而“\\word”作为一个包含有两个“\”的字符串，又需要两个“\”将其转义，因此正确的写法如下所示。

```
re.compile('\\\\\\word')
```

而如果使用原始字符串，则正确的写法如下所示。

```
re.compile(r'\\\\word')
```

5.4 正则表达式对象的属性和方法

正则表达式对象提供了与 `re` 模块中函数类似的字符串操作方法。常用的正则表达式对象的属性和方法可分为以下几种。

5.4.1 匹配和搜索

正则表达式对象的 `match()` 方法用于从字符串开始处进行匹配，或者从指定位置处进

行匹配。要匹配的字符串必须位于开始，或者参数指定的位置才会匹配成功。其原型如下所示。

```
match( string[, pos[, endpos]])
```

其参数含义如下。

- string: 要进行匹配的字符串。
- pos: 可选参数，进行匹配的起始位置。
- endpos: 可选参数，进行匹配的结束位置。

如果匹配成功，match()返回一个 MatchObject 对象实例。与 match()类似，search()方法用于对字符串进行查找，不同的是 search()方法在整个字符串中搜索。如果查找成功，search()将返回一个 MatchObject 对象实例。其原型如下所示。

```
search( string[, pos[, endpos]])
```

其参数含义如下。

- string: 要进行匹配的字符串。
- pos: 可选参数，进行查找的起始位置。
- endpos: 可选参数，进行查找的结束位置。

正则表达式对象的 findall()方法用于在字符串中查找所有符合正则表达式的字符串，并返回这些字符串的列表。如果在正则表达式中使用了组，则返回一个元组。其原型如下所示。

```
findall( string[, pos[, endpos]])
```

其参数含义与 search()方法中的相同。

以下实例使用 RegexObject 对象对字符串进行匹配和搜索。

```
# 导入 re 模块
>>> import re
# 编译正则表达式，“go*d”表示在“g”和“d”之间有任何个以字母“o”开头的单词，如“gd”，“god”，“good”
>>> r = re.compile('go*d')
# 在字符串开始处匹配，没有返回值，表示匹配失败
>>> r.match('Life can be good')
# 从字符串的第 13 个字符开始匹配（字符串从 0 开始），也就是从字母“g”开始，返回 MatchObject 对象实例
>>> r.match('Life can be good',12)
<_sre.SRE_Match object at 0x01122640>
# 在字符串中搜索“go*d”，返回 MatchObject 对象实例，表示字符串中含有“go*d”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011226E8>
# 重新编译，匹配字母“b”和字母“g”之间包含一个字母以及一个空字符的情况
>>> r = re.compile('b.\sg')
# 在字符串中搜索，此处匹配的是“be g”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x01122640>
# 重新编译，匹配任意两个字母后跟一个空字符和字母“g”的情况
```



```
>>> r = re.compile('\w.\sg')
# 在字符串中搜索，此处匹配的是“be g”
>>> r.search('Life can be good')
<_sre.SRE_Match object at 0x011227C8>
# 匹配后边有一个空字符的任意包含两个或者三个字符的单词
>>> r = re.compile('\b\w..?\s')
# 使用 findall() 方法查找
>>> r.findall('Life can be good')
['can ', 'be ']
```

5.4.2 替换

正则表达式对象的 `sub()` 和 `subn()` 方法用于对字符串的替换，其原型分别如下所示。

```
sub(repl, string[, count = 0])
subn(repl, string[, count = 0])
```

其参数含义如下。

- `repl`: 要替换成的内容。
- `string`: 进行内容替换的字符串。
- `count`: 可选参数，最大替换次数。

以下实例将所有以字母“b”开头的单词替换成“*”。

```
>>> import re                                # 导入 re 模块
>>> s = '''Life can be good;                # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('b\w*', re.I)            # 编译正则表达式，忽略大小写
>>> new = r.sub('*', s)                      # 使用 sub() 替换字符
>>> print new                                # 输出结果，可以看到所有以“b”开头的单词都被替换
Life can * good;
Life can * *;
Life is mostly cheerful;
* sometimes sad.
>>> new = r.sub('*', s, 2)                    # 只在字符串中替换两次
>>> print new
Life can * good;
Life can * bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\w*')                  # 重新编译，不忽略大小写
>>> new = r.subn('*', s)                     # 使用 subn() 替换字符，它返回一个元组
>>> print new[0]                             # 输出替换后的字符串，可以看到“But”没有被替换
Life can * good;
Life can * *;
Life is mostly cheerful;
But sometimes sad.
>>> print new[1]                             # 输出替换的次数
```

```

3
>>> new = r.subn('*',s,1)           # 只在字符串中替换一次
>>> print new[0]
Life can * good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> print new[1]
1

```

5.4.3 分割字符串

正则表达式对象的 `split()` 方法用于对字符串进行分割。其原型如下所示。

```
split( string[, maxsplit = 0])
```

其参数含义如下。

- `string`: 要分割的字符串。
- `maxsplit`: 可选参数, 最大分割次数。

以下实例使用 `split()` 方法对字符串进行分割。

```

>>> import re                         # 导入 re 模块
>>> s = '''Life can be good;         # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.'''
>>> r = re.compile('\s')             # 编译匹配空字符的正则表达式
>>> news = r.split(s)                 # 以空字符分割字符串
>>> print news                         # 返回一个列表
['Life', 'can', 'be', 'good;', 'Life', 'can', 'be', 'bad;', 'Life', 'is', 'mostly',
'cheerful;', 'But', 'sometimes', 'sad.']
>>> news = r.split(s,4)               # 只分割 4 次
>>> for new in news:                  # 遍历列表, 输出分割后的字符串
...     print new
...
Life
can
be
good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('b\w*',re.I)      # 编译匹配以字母“b”开头的字符串, 忽略大小写
>>> news = r.split(s)                 # 分割字符串返回列表
>>> print news                         # 输出列表
['Life can ', ' good;\nLife can ', ' ', ';\nLife is mostly cheerful;\n', ' sometimes sad.']
>>> news = r.split(s,1)               # 只分割一次
>>> for new in news:                  # 遍历列表, 输出字符串
...     print new
...

```

```

Life can
good;
Life can be bad;
Life is mostly cheerful;
But sometimes sad.
>>> r = re.compile('\w*e', re.I)           # 编译匹配以字母“e”结尾的字符串，忽略大小写
>>> news = r.split(s)
>>> print news
['', ' can ', ' good;\n', ' can ', ' bad;\n', ' is mostly ', 'rful;\nBut ', 's sad.']

```

5.5 使用组

组允许将正则表达式分解成几个不同的组成部分。在完成匹配或者搜索后，可以使用组编号访问不同部分匹配的内容。

5.5.1 组概述

在正则表达式中以一对圆括号“()”来表示位于其中的内容属于一个组。例如“(re)+”将匹配“rere”、“rerere”等多个“re”重复的情况。组在匹配由不同部分组成的一个整体时非常有用。如电话号码由区号和号码组成，在正则表达式中可以使用两组来进行匹配：一组匹配区号，另一组匹配后边的号码。代码如下所示。

```

>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(\d+)-(\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01107E30>
>>> m.group(1)
'010'
>>> m.group(2)
'87654321'
>>> m.groups()
('010', '87654321')

```

在正则表达式中可以通过使用“(?P<组名>)”为组设置一个名字，通过使用如下所示模式，将第一个组的名字设置为“Area”，将第二个组的名字设置为“No”。

```
r'(?P<Area>\d+)-(?P<No>\d+)'
```

上述的例子可以修改为如下所示的形式。

```

>>> import re
>>> s = 'Phone No. 010-87654321'
>>> r = re.compile(r'(?P<Area>\d+)-(?P<No>\d+)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01122BA8>
>>> m.groupdict()

```



```
{'Area': '010', 'No': '87654321'}
>>> m.group('No')
'87654321'
>>> m.group('Area')
'010'
```

5.5.2 组的扩展语法

除了在组中使用“(?P<>)”来命名组名以外，还可以使用以下几种以“?”开头的扩展语法。

- (?iLmsux): 设置匹配标志，可以是 i、L、m、s、u、x 以及它们的组合。其含义与编译标志相同。
- (?:...): 表示此非一个组。
- (?P=name): 表示在此之前的名为 name 的组。
- (?#...): 表示注释。
- (?=...): 用于正则表达式之后，表示如果“=”后的内容在字符串中出现则匹配，但不返回“=”后的内容。
- (?!...): 用于正则表达式之后，表示如果“!”后的内容在字符串中不出现则匹配，但不返回“!”后的内容。
- (?<=...): 用于正则表达式之前，与(?=...)含义相同。
- (?<!...): 用于正则表达式之前，与(?!...)含义相同。

上述模式的使用如下所示。

```
>>> import re                                # 导入 re 模块
>>> s = '''Life can be good;                # 定义字符串
... Life can be bad;
... Life is mostly cheerful;
... But sometimes sad.
... '''
>>> r = re.compile(r'be(=?\sgood)')          # 编译正则表达式，只匹配其后单词为“good”的“be”
>>> m = r.search(s)                          # 搜索字符串
>>> m                                         # 查看 m
<_sre.SRE_Match object at 0x0111ED40>        # 返回一个 Matchobject 对象实例，表示查找到单词
>>> m.span()                                # 输出匹配到的单词在字符串中的位置
(9, 11)
>>> r.findall(s)                             # 使用 findall() 方法输出所有匹配的单词
['be']
>>> r = re.compile('be')                    # 重新编译正则表达式，匹配单词“be”
>>> r.findall(s)                             # 使用 findall() 方法输出所有匹配的单词
['be', 'be']
>>> r = re.compile(r'be(?!\sgood)')          # 匹配之后单词不为“good”的“be”
>>> m = r.search(s)                          # 搜索字符串
>>> m
```

第5章 正则表达式

```

<_sre.SRE_Match object at 0x010DAAD8>
>>> m.span()
(27, 29)
>>> r = re.compile(r'(?can\s)be(\sgood)')
>>> m = r.search(s)
>>> m
<_sre.SRE_Match object at 0x01112660>
>>> m.groups()
(' good',)
>>> m.group(1)
' good'
>>> r = re.compile(r'(?P<first>\w)(?P=first)')
>>> r.findall(s)
['o', 'e']
>>> r = re.compile(r'(?<=can\s)b\w*\b')
>>> r.findall(s)
['be', 'be']
>>> r = re.compile(r'(?<!can\s)b\w*\b')
>>> r.findall(s)
['bad']
>>> r = re.compile(r'(?<!can\s)(?i)b\w*\b')
>>> r.findall(s)
['bad', 'But']

```

返回一个 Matchobject 对象实例, 表示查找到单词
输出匹配到的单词在字符串中的位置

使用组来匹配 “be good”

使用 groups() 方法输出组

使用组编号输出组

使用组名重复, 此处匹配具有两个重复字母的单词
输出匹配到的字母

匹配以字母 “b” 开头位于 “can” 之后的单词
输出匹配到的单词

匹配以字母 “b” 开头不位于 “can” 之后的单词

重新编译, 忽略大小写

5.6 Match 对象

Match 对象实例是由正则表达式对象的 `match`, 以及 `search` 方法在匹配成功后返回的。Match 对象有以下常用的方法和属性, 用于对匹配成功的正则表达式进行处理。

5.6.1 使用 Match 对象处理组

`group()`、`groups()`以及 `groupdict()`方法都是处理在正则表达式中使用 “()” 分组的情况。不同的是, `group()`的返回值为字符串, 当传递多个参数时其返回值为元组。`groups()`的返回值为元组。`groupdict()`的返回值为字典。其原型分别如下所示。

```

group( [group1, ...])
groups( [default])
groupdict( [default])

```

对于 `group()`而言, 其参数为分组的编号。如果向 `group()`传递多个参数, 则其返回各个参数所对应的字符串组成的元组。对于 `groups()`和 `groupdict()`一般不需要向其传递参数。以下实例使用上述 3 种方法对字符串进行操作。

```

>>> import re
>>> s = '''Life can be dreams,
... Life can be great thoughts;
... Life can mean a person,

```

导入 re 模块
定义字符串

```

... Sitting in a court.'''
>>> r = re.compile('\\b(?P<first>\\w+)a(\\w+)\\b')
# 编译正则表达式，匹配所有包含字母“a”的单词
>>> m = r.search(s)
# 从头开始搜索，search() 返回搜索到的第一个单词
>>> m.groupdict()
# 使用 groupdict() 输出字典
{'first': 'c'}
>>> m.groups()
# 使用 groups() 输出元组
('c', 'n')
>>> m = r.search(s, 9)
# 从指定位置开始重新搜索
>>> m.group()
# 输出匹配到的字符串
'dreams'
>>> m.group(1)
# 输出第一对圆括号中的内容，即字母“a”之前部分
'dre'
>>> m.group(2)
# 输出第二对圆括号中的内容，即字母“a”之后部分
'ms'
>>> m.group(1, 2)
# 全部输出，返回一个元组
('dre', 'ms')
>>> m.groupdict()
# 使用 groupdict() 输出字典
{'first': 'dre'}
>>> m.groups()
# 使用 groups() 输出元组
('dre', 'ms')

```

5.6.2 使用 Match 对象处理索引

start()、end()以及 span()方法返回所匹配的子字符串的索引。其原型分别如下所示。

```

start([groupid=0])
end([groupid=0])
span([groupid=0])

```

其参数含义相同，groupid 为可选参数，即分组编号。如果不向其传递参数，则返回整个子字符串的索引。start()方法返回子字符串或者组的起始位置索引。end()方法返回子字符串或者组的结束位置索引。而 span()方法则以元组的形式返回以上两者。其使用方法如下所示。

```

>>> import re
# 导入 re 模块
>>> s = '''Life can be dreams,
# 定义字符串
... Life can be great thoughts;
... Life can mean a person,
... Sitting in a court.'''
r = re.compile('\\b(?P<first>\\w+)a(\\w+)\\b')
# 编译正则表达式匹配含有字母“a”的单词
m = r.search(s, 9)
# 从字符串中第 10 个字符开始搜索
>>> m.start()
# 输出匹配到的子字符串的起始位置
12
>>> m.start(1)
# 输出第一组的起始位置
12
>>> m.start(2)
# 输出第二组的起始位置
16
>>> m.end(1)
# 输出第一组的子字符串的结束位置
15

```



```

>>> m.end()
18
>>> m.span()
(12, 18)
>>> m.span(2)
(16, 18)

```

输出子字符串的结束位置

输出子字符串的开始和结束位置

输出第二组子字符串的开始和结束位置

5.7 使用正则表达式处理文件

正则表达式是处理文本文件的强有力的工具。本节给出一个简单的使用正则表达式处理 Python 脚本中的函数和变量的例子。

在 Python 脚本中，函数定义必须以“def”开头，因此处理函数的过程相当简单。为了代码简洁，此处假设脚本编写规范“def”后跟一个空格，然后是函数名，接着就是参数。没有考虑使用多个空格的情况。

而 Python 脚本中的变量不好处理，因为变量一般不需要事先声明，往往都是直接赋值，因此在脚本中首先处理了变量直接赋值的情况。通过匹配单词后接“=”的情况查找变量名。同样，为了代码简洁，仅考虑比较规范整洁的写法，变量名与“=”之间有一空格。另外，还有一类变量是在 for 循环语句中直接使用的，因此脚本中又特别处理了 for 循环的情况。为了使代码简洁，脚本并没有处理变量名重复的情况。整个脚本的代码如下所示。

```

# -*- coding:utf-8 -*-
# file : GetFunction.py
#
import re
import sys
def DealWithFunc(s):
    r = re.compile(r'''
        (?<=def\s)
        \w+
        \(. * ? \)
        (?=:)
    ''', re.X)
    return r.findall(s)
def DealWithVar(s):
    vars = []
    r = re.compile(r'''
        \b
        \w+
        (?=\s=)
    ''', re.X)
    vars.extend(r.findall(s))
    r = re.compile(r'''
        (?<=for\s)

```

前边必须含有 def 且 def 后跟一个空格

匹配函数名

匹配参数

后边必须跟一个“:”

设置编译选项，忽略模式中的注释

定义一个列表，因为这里分两种情况处理

匹配单词开始

匹配变量名

处理未为变量赋值的情况

处理变量位于 for 语句中的情况

```

        \w+                # 匹配变量名
        \s                 # 匹配空格
        (?=in)             # 匹配 in
        '',re.X)           # 设置编译选项，忽略模式中的注释
    vars.extend(r.findall(s))
    return vars
# 判断命令行是否有输入，没有则要求输入要处理的文件
if len(sys.argv) == 1:
    sour = raw_input('请输入要处理的文件路径')
else:
    sour = sys.argv[1]
file = open(sour)          # 打开文件
s = file.readlines()       # 将文件内容以行读入 s 中
file.close()              # 关闭文件
print '*****'
print sour, '中的函数有: '
print '*****'
i = 0                      # i 为函数所在的行号
# 循环处理每一行，匹配其中的函数，并输出函数所在的行号，以及函数的原型
for line in s:
    i = i + 1
    function = DealWithFunc(line)
    if len(function) == 1:
        print 'Line: ', i, '\t', function[0]
print '*****'
print sour, '中的变量有: '
print '*****'
i = 0                      # 此处 i 为变量所在的行号
# 循环处理每一行，匹配其中的变量，输出变量所在的行号，以及变量名
for line in s:
    i = i + 1
    var = DealWithVar(line)
    if len(var) == 1:
        print 'Line: ', i, '\t', var[0]

```

运行脚本后，让其处理自身，脚本输出如下所示。

```

*****
GetFunction.py 中的函数有:
*****
Line: 6      DealWithFunc(s)
Line: 14     DealWithVar(s)
*****
GetFunction.py 中的变量有:
*****
Line: 7      r
Line: 15     vars
Line: 16     r
Line: 22     r
Line: 32     sour
Line: 34     sour

```

```
Line: 35      file
Line: 36      s
Line: 41      i
Line: 43      line
Line: 44      i
Line: 45      function
Line: 51      i
Line: 53      line
Line: 54      i
Line: 55      var
```


第 6 章 面向对象的 Python

Python 是一种面向对象的编程语言，但是 Python 与 C++ 一样，还支持面向过程的程序设计。在 Python 中完全可以使用函数、模块等方式来完成工作。但当使用 Python 编写一个较为庞大的项目时，则应该考虑使用面向对象的方法，以便更好地对项目进行管理。

6.1 概述

面向对象程序设计 (Object Oriented Programming) 简称 OOP，是与面向过程的程序设计不同的另一种编程架构。在此之前的例子大都是面向过程的，而非面向对象的。

6.1.1 Python 中的面向对象的思想

面向对象程序设计是从 20 世纪 90 年代开始流行的一种编程方法，强调对象的“抽象”、“封装”、“继承”、“多态”。面向对象程序设计方法的基本思想是将任何事物都当作对象，是其所属对象类的一个实例。对于复杂的对象则将其划分成简单的对象，由这些简单的对象以某种方式组合而形成复杂的对象。每一个对象都有其相对应的对象类。属于同一对象类的对象具有相同的属性以及操作方法等。

对象以对象类的形式将其内部的数据或者方法封装。对象与对象之间只是相互传递数据，而不能访问其他对象的内部。对象的内部相对于其他对象来说是不可见的。不同的对象类之间可以通过继承的形式来拥有其他对象的属性和方法等，而形成父子关系。面向对象程序设计方法的基本过程如下。

- (1) 确定对象及其属性和方法等。
- (2) 分析对象之间的联系确定其通信机制。
- (3) 将具有共同特征的对象抽象为对象类。
- (4) 设计、实现类，并确定类相互间的继承关系。
- (5) 创建对象实例，实现对象间的相互联系。

例如，可以将人作为一个对象类。每一个具体的人，如张三，则是一个对象实例。每个

人都具有名字、性别、年龄、身高等，则可以将这些抽象为对象类的属性。

Python 完全采用了面向对象程序设计的思想。在 Python 中可以使用类建立一个对象模型，以及对象所拥有的属性和方法等。该模型能够较好地反映事物的本质，以及其相互之间的关系，其本质是更接近于人类认知事物所采用的计算模型。

Python 是真正面向对象的脚本语言，虽然其与 C++ 的类机制有所区别，但 Python 保证对类的最重要功能的支持，例如类的继承、基类的重载等。

在 Python 中，对象概念比较广泛，对象不一定非得是类的实例。Python 的内置数据类型如字符串、列表、字典等，它们都不是类，但却多少具有和类相似的语法。例如，使用“.”操作符来使用内置类型的某些方法。

6.1.2 类和对象

类是面向对象程序设计的基础。类具有抽象性、封装性、继承性和多态性。

- 类的抽象性是指类是对具有共同方法和属性的一类对象的描述。
- 类的封装性是指类将属性和方法封装，对于外部其是不可见的，只有通过类提供的接口才能与属于类的实例对象进行信息交换。
- 类的继承性是指类可以由已有的类派生。派生出来的类拥有父类的方法和属性等。
- 类的多态性是指类可以根据不同的参数类型调用不同的方法。同一个方法可以处理不同类型的参数。实际上 Python 的内部已经很好地实现了多态。在 Python 中使用类不需要考虑太多不同类型数据之间的处理。

每个类都具有自己的属性和方法。类的属性实际上就是类内部的变量。而类的方法，则是在类内部定义的函数。

对象是具体的事物，是实例化后的类。每个对象的属性值可能不一样，但所有由同一类实例化得来的对象都拥有共同的属性和方法。在程序中由类实例化生成对象，然后使用对象的方法进行操作，完成任务。一个类可以实例化生成多个对象。类与对象的关系如图 6-1 所示。

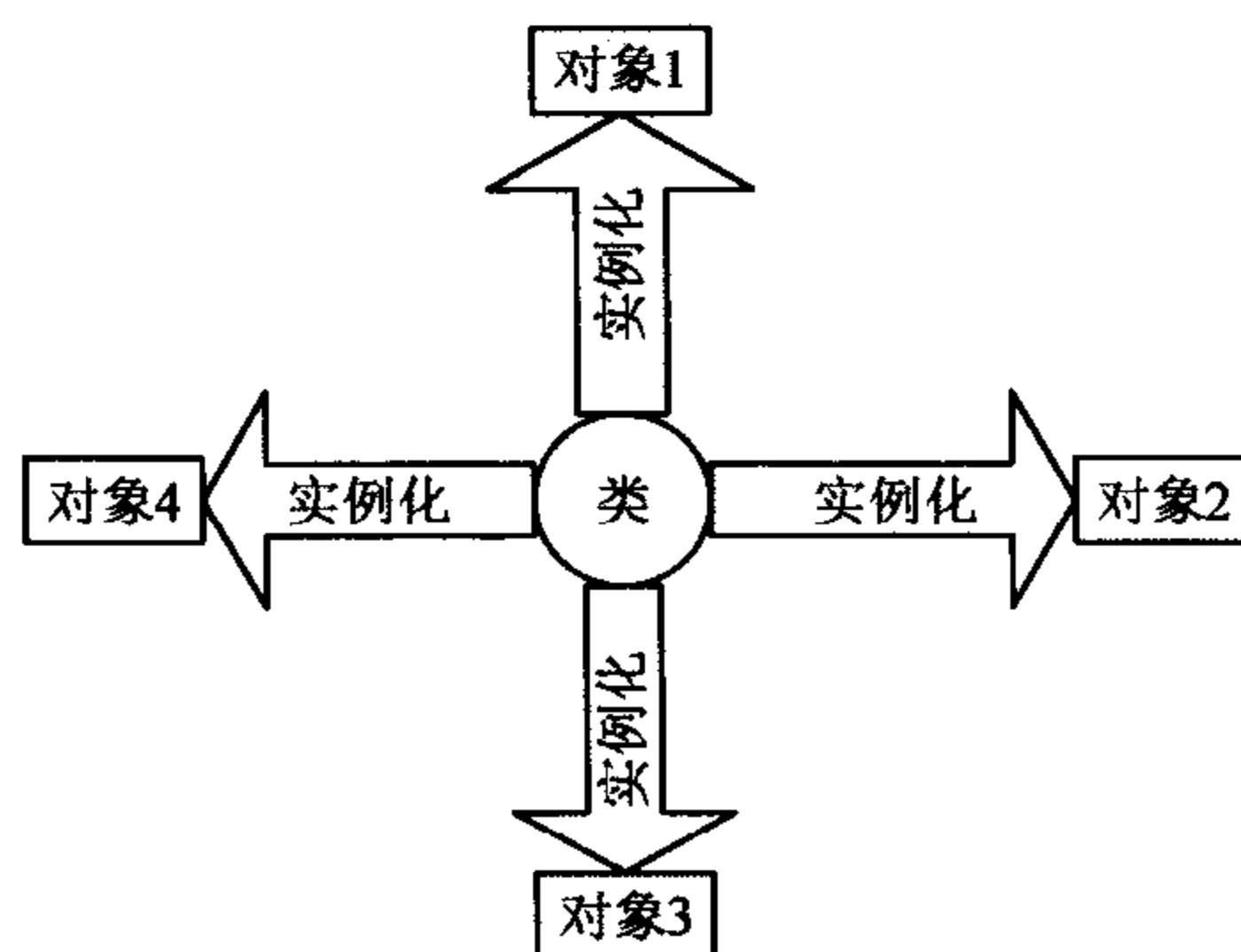


图 6-1 类与对象的关系

6.2 类的基础

由于 Python 对面向对象程序设计的良好支持，在 Python 中定义和使用类并不复杂。类的定义和使用与函数的定义和使用有很多相似的地方。

6.2.1 类的定义

在 Python 中类的定义与函数的定义类似，不同的是，类的定义使用关键字“class”。与函数定义相同，在定义类的时候也要使用缩进以表示缩进的语句属于该类。一般类的定义形式如下所示。

```
class <类名>:
    <语句 1>
    <语句 2>
    ...
    <语句 3>
```

与函数定义一样，在使用类之前必须先定义类。类的定义一般放在脚本的头部。在 Python 中也可以在 if 语句的分支中或者函数定义中定义类。以下实例定义了一个 human 类，并定义了相关属性。

```
>>> class human:                # 定义 human 类
...     age = 0                  # 定义 age 属性
...     sex = ''                 # 定义 sex 属性
...     height = 0              # 定义 height 属性
...     weight = 0              # 定义 weight 属性
...     name = ''               # 定义 name 属性
```

类还可以通过继承的形式获得。通过类继承来定义类的基本形式如下所示。

```
class <类名>(<父类名>):
    <语句 1>
    <语句 2>
    ...
    <语句 3>
```

其中圆括号中的父类名就是要继承的类。关于继承将在后边的章节中讲解，此处给出一个简单的例子。如下所示代码通过继承 human 类来生成一个新类。

```
>>> class student(human):        # 通过继承 human 类创建 student 类
...     school = ''              # 定义新属性 school
...     number = 0               # 定义新属性 number
...     grade = 0               # 定义新属性 grade
... 
```

上述通过 human 继承而来的 student 类具有 human 类的属性，并且又为 student 类定义了其他的属性。

类定义后就产生了一个名字空间，与函数类似。在类内部使用的属性，相当于函数中的变量名，还可以在类的外部继续使用。类的内部与函数的内部一样，相当于一个局部作用域。

不同类的内部也可以使用相同的属性名。

6.2.2 类的使用

类在定义后必须先实例化才能使用。类的实例化与函数调用类似，只要使用类名加圆括号的形式就可以实例化一个类。类实例化以后会生成一个对象。一个类可以实例化多个对象，对象与对象之间并不相互影响。类实例化以后可以使用其属性和方法等。以下实例首先定义一个 book 类，然后将其实例化。

```
>>> class book:
...     author = ''
...     name = ''
...     pages = 0
...     price = 0
...     press = ''
...
# 定义 book 类
# 定义 author 属性
# 定义 name 属性
# 定义 pages 属性
# 定义 price 属性

>>> a = book()
# book 类实例化

>>> a
# 查看对象 a
<class _main_.book at 0x011205A0>

>>> a.author
# 访问 author 属性
''

>>> a.pages
# 访问 pages 属性
0

>>> a.price
# 访问 price 属性
0

>>> a.author = 'Tom'
# 设置 author 属性

>>> a.pages = 300
# 设置 pages 属性

>>> a.price = 25
# 设置 price 属性

>>> a.author
# 重新访问 author 属性
'Tom'

>>> a.pages
# 重新访问 pages 属性
300

>>> a.price
# 重新访问 price 属性
25

>>> b = book()
# 将 book 实例化生成 b 对象

>>> b.author
# 访问 b 对象的 author 属性
''

>>> b.price
# 访问 b 对象的 price 属性
0

>>> b.author = 'Jack'
# 设置 b 对象的 author 属性

>>> b.author
'Jack'

>>> b.price = 15
# 设置 b 对象的 price 属性

>>> b.price
15

>>> a.price
# a 对象的 price 属性并没有改变
25

>>> a.author
# a 对象的 author 属性也没有改变
'Tom'
```

```

>>> b.pages          # 访问 b 对象的 pages 属性
0
>>> a.pages          # 访问 a 对象的 pages 属性
300

```

上述例子只定义了类的属性，并在类实例化以后重新设置其属性，从上例可以看出类的实例化相当于调用一个函数，这个函数就是类。函数返回一个类的实例对象，返回后的对象就具有了类所定义的属性。上述例子生成了两个 book 实例对象，可以看到，设置其中一个对象的属性，并不影响另一个对象的属性。

在 Python 中需要注意的是，虽然类首先需要实例化，然后才能使用其属性。而实际上当创建一个类以后就可以通过类名访问其属性。如果直接使用类名修改其属性，那么将影响已经实例化的对象。代码如下所示。

```

>>> class A:          # 定义类 A
...     name = 'A'    # 定义 name 属性将其赋值为 'A'
...     num = 2       # 定义 num 属性将其赋值为 2
...
>>> A.name           # 直接使用类名访问类的属性
'A'
>>> A.num            # 直接使用类名访问类的属性
2
>>> a = A()          # 生成 a 对象
>>> a.name           # 查看 a 的 name 属性
'A'
>>> b = A()          # 生成 b 对象
>>> b.name           # 查看 b 的 name 属性
'A'
>>> A.name = 'B'     # 使用类名修改 name 属性
>>> a.name           # a 对象的 name 属性被修改
'B'
>>> b.name           # b 对象的 name 属性也被修改
'B'

```

6.3 类的属性和方法

每一个类都具有自己的属性和方法。属性和方法是面向对象程序设计所独有的概念。属性是类所封装的数据，而方法则是类对数据进行的操作。

6.3.1 类的属性

在上一节中已经简单地定义和使用了类的属性。类的属性实际上是类内部的变量。上一节的例子使用了类的属性，确切地说，称作类的公有属性。在上一节的例子中在类的外部设置其属性的值，在某些情况下可能不希望在类的外部对其属性进行操作，此时就可以使用类的私有属性。

数据保护是面向对象程序设计所特有的，在面向过程的程序设计中并没有数据保护的概念。在 Python 中与 C++ 不同，在类的内部声明一个私有成员不需要使用 `private` 关键字。在 Python 中，是通过类中属性的命名形式来表示类属性是公有还是私有的。类中的私有属性是不能在类的外部进行操作的，这便起到了对属性的保护作用。

在 Python 中，如果类中的属性是以两条下划线开始的话，则该属性为类的私有属性，不能在类的外部被使用或者访问。如下所示为一个私有属性的命名形式。

```
_priavte_attrs          # 以双下划线开始

如果在类内部的方法中使用类的私有属性，则应该以如下的方式调用。
self._priavte_attrs      # 应该在私有属性名前加上“self.”

以下代码修改上一节中的 book 类，将其部分属性改为私有属性。

>>> class book:          # 定义一个类
...     _author = ''      # 类的私有属性
...     _name = ''        # 类的私有属性
...     _page = 0         # 类的私有属性
...     price = 0         # 类的公有属性
...     _press = ''       # 类的私有属性
...
>>> a = book()           # 实例化 book 类生成 a 对象
>>> a._author            # 试图访问对象的 _author 私有属性，结果导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: book instance has no attribute '_author'
>>> a.price              # 访问对象的 price 公有属性
0
>>> a.price = 20         # 修改 price 属性
>>> a.price              # price 属性改变了
20
>>> a._name              # 试图访问对象的 _name 私有属性，结果导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: book instance has no attribute '_name'
>>> a._page              # 试图访问对象的 _page 私有属性，结果导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: book instance has no attribute '_page'
```

可以看到，在类定义的时候，凡是以两条下划线开始的属性不能在类的外部访问，当然也不能修改。如果要修改类的私有属性值或者获取其值，可以通过使用类提供的方法来完成。

6.3.2 类的方法

类的方法实际上就是类内部使用 `def` 关键字定义的函数。定义类的方法与定义一个函数基本相同，在类的方法中同样也要使用缩进。

1. 定义类的方法

在类的内部使用 `def` 关键字可以为类定义一个方法。与函数定义不同的是，类的方法必须包含参数“`self`”，且“`self`”必须为第一个参数。如下代码为上一节中的 `book` 类添加一个 `show` 方法和 `setname` 方法。

```
>>> class book:
...     _author = ''
...     _name = ''
...     _page = 0
...     price = 0
...     _press = ''
...     def show(self):
...         print self._author
...         print self._name
...     def setname(self, name):
...         self._name = name
...
>>> a = book()
>>> a.show()

>>>
>>> a.setname('Tom')
>>> a.show()
```

定义 book 类
类的私有属性
类的私有属性
类的私有属性
类的公有属性
类的私有属性
定义类的 show 方法，其参数必须为 self
输出类的私有属性
输出类的私有属性
定义类的 setname 方法，其有两个参数
设置类的 _name 私有属性
生成类的实例
调用 show 方法，输出为空，因为其私有属性都为空
调用 setname 方法，但只向其传递一个参数
调用 show 方法

Tom

与类的属性相同，类的方法也可以是类私有的，类的私有方法不能在类的外部调用。和类的私有属性命名相同，类的私有方法名也要以两条下划线开始。类的私有方法只能在类的内部调用，而不能在类的外部调用。另外，在类的内部调用其私有方法，要使用“`self.私有方法名`”的形式。如下所示代码为 `book` 类添加一个名为 `check` 的私有方法，并且修改 `show()` 方法。

```
>>> class book:
...     _author = ''
...     _name = ''
...     _page = 0
...     price = 0
...     _press = ''
...     def _check(self, item):
...         if item == '':
...             return 0
...         else:
...             return 1
...     def show(self):
...         if self._check(self._author):
...             print self._author
...         else:
```

定义 book 类
类的私有属性
类的私有属性
类的私有属性
类的公有属性
类的私有属性
定义 _check 私有方法
判断 item 是否为空
为空，则返回 0
否则返回 1
修改 show 方法
判断 _author 私有属性是否为空
不为空，则输出其值

```
...         print 'No value'           # 否则输出 "No value"
...     if self._check(self._name):    # 判断私有属性 _check 是否为空
...         print self._name          # 不为空, 则输出其值
...     else:
...         print 'No value'           # 否则输出 "No value"
... def setname(self,name):            # 定义 setname 方法
...     self._name = name
...
>>> a = book()                       # 类的实例化
>>> a.show()                          # 调用 show 方法
No value
No value
>>> a.setname('Tom')                  # 调用 setname 方法
>>> a.show()                          # 重新调用 show 方法, 看到输出值已经改变
No value
Tom
>>> a._check()                        # 调用类的私有方法, 结果出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: book instance has no attribute '_check'
```

2. 类的专有方法 (Special Methods)

在 Python 中有一类以两条下划线开始并且以两条下划线结束的类的方法, 称之为类的专有方法。专有方法是针对类的特殊操作。例如, 在类实例化时将调用_init_方法。部分类的专有方法如表 6-1 所示。

表 6-1 类的专有方法

方 法 名	描 述	方 法 名	描 述
<code>_init_</code>	构造函数, 生成对象时调用	<code>_repr_</code>	打印、转换
<code>_del_</code>	析构函数, 释放对象时调用	<code>_setitem_</code>	按照索引赋值
<code>_add_</code>	加运算	<code>_getitem_</code>	按照索引获取值
<code>_mul_</code>	乘运算	<code>_len_</code>	获得长度
<code>_cmp_</code>	比较运算	<code>_call_</code>	函数调用

以下代码修改 book 类, 使用_init_方法为对象的属性赋初始值。

```
>>> class book:                      # 定义 book 类
...     _author = ''                 # 类的私有属性
...     _name = ''                  # 类的私有属性
...     _page = 0                    # 类的私有属性
...     price = 0                    # 类的公有属性
...     _press = ''                 # 类的私有属性
...     def _check(self,item):       # 定义 _check 私有方法
...         if item == '':           # 判断 item 是否为空
...             return 0              # 为空, 则返回 0
...         else:
...             return 1              # 否则返回 1
```

```

... def show(self):
...     if self._check(self._author):
...         print self._author
...     else:
...         print 'No value'
...     if self._check(self._name):
...         print self._name
...     else:
...         print 'No value'
... def setname(self,name):
...     self._name = name
... def _init_(self,author,name):
...     self._author = author
...     self._name = name
...
>>> a = book('Tom','A Wonderfull Book')
>>> a.show()
Tom
A Wonderfull Book
>>> a.setname('About Jack')
>>> a.show()
Tom
About Jack

```

修改 show 方法
判断 _author 私有属性是否为空
不为空，则输出其值
否则输出 “No value”
判断私有属性 _check 是否为空
不为空，则输出其值
否则输出 “No value”
定义 setname 方法
使用 _init_ 方法，为 _author 及 _name 赋初值
实例化，为 _author 及 _name 赋初值
调用 show 方法
重新设置 _name 私有属性
调用 show 方法

6.4 类的继承

一个新类可以通过继承来获得已有类的方法以及属性等。通过继承而来的类也可以自己定义新的方法或者属性。

6.4.1 通过继承创建类

在类的定义中已经提到如何通过继承来获得一个新类。新类可以继承父类的公有属性和公有方法，但是不能继承父类的私有属性和私有方法。如下所示代码通过继承 book 类来创建一个名为 student 的类。

```

>>> class book:
...     _author = ''
...     _name = ''
...     _page = 0
...     price = 0
...     _press = ''
...     def _check(self,item):
...         if item == '':
...             return 0
...         else:
...             return 1
...     def show(self):

```

定义 book 类
类的私有属性
类的私有属性
类的私有属性
类的公有属性
类的私有属性
定义 _check 私有方法
判断 item 是否为空
为空，则返回 0
否则返回 1
修改 show 方法

第6章 面向对象的Python

```

...     if self._check(self._author):          # 判断_author 私有属性是否为空
...         print self._author                # 不为空, 则输出其值
...     else:
...         print 'No value'                  # 否则输出 "No value"
...     if self._check(self._name):           # 判断私有属性_check 是否为空
...         print self._name                  # 不为空, 则输出其值
...     else:
...         print 'No value'                  # 否则输出 "No value"
... def setname(self,name):                    # 定义 setname 方法
...     self._name = name
... def _init_(self,author,name):              # 使用_init_方法, 为_author 及_name 赋初值
...     self._author = author
...     self._name = name
...
>>> class student(book):                      # 通过继承创建 student 类
...     _class = ''
...     _grade = ''
...     _sname = ''
...     def showinfo(self):                    # 定义一个新的 showinfo 方法
...         self.show()                       # 此处仅调用 book 类的 show 方法
...
>>> b = student('Jack','Big Book')            # student 类继承了 book 类的_init_方法
>>> b.showinfo()                              # 调用 student 类的 showinfo 方法
Jack
Big Book
>>> b.show()                                  # 调用 book 类的 show 方法
Jack
Big Book

```

如果在定义类时试图使用父类的私有属性或者私有方法将会导致错误, 如下所示。

```

>>> class student(book):
...     def showall(self):
...         # 此处调用了父类的_check方法, 且使用了父类的_name 私有属性
...         if self._check(self._name):
...             print self._name
...         else:
...             print 'No Value'
...
>>> c = student('Tom','Big Book')
>>> c.showall()                                # 调用 showall 方法时出错
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "<interactive input>", line 3, in showall
AttributeError: student instance has no attribute '_student_check'

```

6.4.2 多重继承

多重继承是指创建的类同时拥有几个类的属性和方法。多重继承与单重继承不同的是在类名后边的圆括号中包含多个父类名, 父类名之间以逗号隔开。通过多重继承创建一个新类

的一般形式如下所示。

```
>>> class 新类名(父类 1, 父类 2, ..., 父类 n):
    <语句 1>
    <语句 2>
    ...
    <语句 3>
```

使用多重继承需要注意圆括号中父类名字的顺序。如果父类中有相同的方法名，而在类中使用时未指定父类名，Python 解释器将从左至右搜索。如下代码首先定义两个类，然后通过多重继承创建一个新类。

```
>>> class A:
...     name = 'A'
...     _num = 1
...     def show(self):
...         print self.name
...         print self._num
...     def setnum(self, num):
...         self._num = num
...
>>> class B:
...     nameb = 'B'
...     _numb = 2
...     def show(self):
...         print self.nameb
...         print self._numb
...     def setname(self, name):
...         self.nameb = name
...
>>> class C(A, B):
...     def showall(self):
...         print self.name
...         print self.nameb
...
>>> c = C()
>>> c.showall()
A
B
>>> c.show()
A
1
>>> c.setnum(3)
>>> c.show()
A
3
>>> c.setname('D')
>>> c.showall()
A
D
```

定义类 A
定义其 name 属性
定义其 _num 属性
定义 show 方法

定义 setnum 方法以设置 _num 属性

定义类 B
定义 nameb 属性
定义 _numb 属性
定义 show 方法

定义 setname 方法

通过多重继承创建 C 类
定义 showall 方法

实例化 C 类生成 c 对象
调用 showall 方法

调用 show 方法，此处调用的是 A 类的 show 方法

调用 setnum 方法，即 A 类的 setnum 方法

调用 setname 方法，即 B 类的 setname 方法

如果需要在类 C 中使用类 B 的 show 方法，可以按如下所示代码修改 C 类。

```
>>> class C(A,B):                                # 通过多重继承创建 C 类
...     def showall(self):
...         print self.name
...         print self.nameb
...     show = B.show                             # 这里表明 show 方法为 B 类的 show 方法
...
>>> c = C()
>>> c.show()                                     # 调用 show 方法，此处调用的是 B 类的 show 方法
B
2
```

6.5 重载

重载允许通过继承而创建的类重新定义父类的方法。不仅可以重载方法，而且还可以重载运算符，例如“+”、“*”等，以适用自创建的类。

6.5.1 方法重载

通过继承而创建的类，其父类的方法不一定能满足类的需要。新类实际上只是修改部分功能，为了避免命名函数的麻烦，可以使用方法重载来解决。或者，新类需要重新初始化，此时就可以重载 `_init_` 方法来实现。

方法的重载实际上就是在类中使用 `def` 关键字重载父类的方法。如果重载父类中的方法，但又需要在类中先使用父类的该方法，可以使用父类名加“.”加方法名的形式调用。例如重载 `_init_` 方法时，而父类也需要使用 `_init_` 方法，则可以在 `_init_` 前加上父类名来调用该方法。如下代码首先定义一个父类，然后通过继承创建一个新类并且重载父类的方法。

```
>>> class human:                                # 定义 human 类
...     _age = 0                                 # 定义 _age 属性
...     _sex = ''                               # 定义 _sex 属性
...     _height = 0                             # 定义 _height 属性
...     _weight = 0                             # 定义 _weight 属性
...     name = ''                               # 定义 name 属性
...     def _init_(self,age,sex,height,weight): # 重载 _init_ 方法
...         self._age = age                     # 初始化 _age 属性
...         self._sex = sex                     # 初始化 _sex 属性
...         self._height = height               # 初始化 _height 属性
...         self._weight = weight               # 初始化 _weight 属性
...     def setname(self,name):                 # 定义 setname 方法
...         self.name = name
...     def show(self):                         # 定义 show 方法
...         print self.name
...         print self._age
...         print self._sex
...         print self._height
```

```
...     print self._weight
...
>>> class student(human):                # 通过继承 human 类生成 student 类
...     _classes = 0                      # 定义 _classes 属性
...     _grade = 0                        # 定义 _grade 属性
...     _num = 0                          # 定义 _num 属性
...     def _init_(self,classes,grade,num,age,sex,height,weight): # 重载 _init_ 方法
...         self._classes = classes
...         self._grade = grade
...         self._num = num
...         human._init_(self,age,sex,height,weight)
...                                     # 调用 human 类的 _init_ 方法初始化 human 类的属性
...     def show(self):                  # 重载 show 方法
...         human.show(self)             # 调用 human 类的 show 方法
...         print self._classes
...         print self._grade
...         print self._num
...
>>> a = student(12,3,20070305,19,'male',175,65) # 实例化生成 a 对象
>>> a.setname('Tom')                          # 调用 setname 方法
>>> a.show()                                  # 调用 show 方法, 即重载后的 show 方法输出属性
Tom
19
male
175
65
12
3
20070305
```

6.5.2 运算符重载

在 Python 中运算符重载不需要像在 C++ 中那样使用 operator 关键字。由于在 Python 中，运算符都有其相对应的函数。在类中，运算符对应类的专有方法。因此运算符的重载实际上是对运算符对应的专有方法的重载。部分运算符和类的专有方法名对照如表 6-2 所示。

表 6-2 运算符与专有方法名对照表

运 算 符	专 有 方 法	运 算 符	专 有 方 法
+	<code>_add_</code>	/	<code>_div_</code>
-	<code>_sub_</code>	%	<code>_mod_</code>
*	<code>_mul_</code>	**	<code>_pow_</code>

```
>>> class MyList:                        # 定义 MyList 类
...     _mylist = []                     # 定义 _mylist 属性
...     def _init_(self, *args):         # 重载 _init_ 方法
...         self._mylist = []           # 此处相当于 _mylist 初始化, 避免多个实例对象数据混合
...         for arg in args:
```

第6章 面向对象的Python

```

...         self._mylist.append(arg)
... def _add_(self,n):                                # 重载 "+" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] + n
... def _sub_(self,n):                                # 重载 "-" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] - n
... def _mul_(self,n):                                # 重载 "*" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] * n
... def _div_(self,n):                                # 重载 "/" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] / n
... def _mod_(self,n):                                # 重载 "%" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] % n
... def _pow_(self,n):                                # 重载 "**" 运算符
...     for i in range(0,len(self._mylist)):
...         self._mylist[i] = self._mylist[i] ** n
... def _len_(self):                                  # 重载 len 方法
...     return len(self._mylist)
... def show(self):                                   # 定义 show 方法
...     print self._mylist
...
>>> l = MyList(1,2,3,4,5)                            # 实例化生成 l 对象
>>> l.show()                                           # 调用 show 方法
[1, 2, 3, 4, 5]
>>> l + 5                                              # 此处将调用 _add_ 方法
>>> l.show()                                           # 调用 show 方法
[6, 7, 8, 9, 10]
>>> l - 3                                              # 此处将调用 _sub_ 方法
>>> l.show()                                           # 调用 show 方法
[3, 4, 5, 6, 7]
>>> l * 6                                              # 此处将调用 _mul_ 方法
>>> l.show()                                           # 调用 show 方法
[18, 24, 30, 36, 42]
>>> l / 3                                              # 此处将调用 _div_ 方法
>>> l.show()                                           # 调用 show 方法
[6, 8, 10, 12, 14]
>>> l % 3                                              # 此处将调用 _mod_ 方法
>>> l.show()                                           # 调用 show 方法
[0, 2, 1, 0, 2]
>>> l ** 3                                             # 此处将调用 _pow_ 方法
>>> l.show()                                           # 调用 show 方法
[0, 8, 1, 0, 8]
>>> len(l)                                             # 此处将调用 _len_ 方法
5
>>> b = MyList(2,3,5,6,7,8,9)                        # 实例化生成 b 对象
>>> len(b)                                             # 此处将调用 _len_ 方法

```



```

7
>>> b.show()                                # 调用 show 方法
[2, 3, 5, 6, 7, 8, 9]
>>> b - 5                                    # 此处将调用 _sub_ 方法
>>> b.show()                                # 调用 show 方法
[-3, -2, 0, 1, 2, 3, 4]
>>> l.show()                                # 调用 l 的 show 方法, 比较 l 和 b 的值
[0, 8, 1, 0, 8]

```

6.6 模块中的类

类与函数一样,也可以写到模块中。在其他的脚本中可以通过导入模块名使用定义的类。模块中类的使用方式与模块中的函数类似。实际上可以将模块中的类当作函数一样使用。将上一节中定义的 MyList 类整理一下保存在 MyList.py 中。代码如下所示。

```

# -*- coding:utf-8 -*-
# file: MyList.py
#
class MyList:                                # 定义 MyList 类
    _mylist = []                             # 定义 _mylist 属性
    def _init_(self, *args):                 # 重载 _init_ 方法
        self._mylist = []                   # 此处相当于 _mylist 初始化, 避免多个实例对象数据混合
        for arg in args:
            self._mylist.append(arg)
    def _add_(self,n):                        # 重载 "+" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] + n
    def _sub_(self,n):                        # 重载 "-" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] - n
    def _mul_(self,n):                        # 重载 "*" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] * n
    def _div_(self,n):                        # 重载 "/" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] / n
    def _mod_(self,n):                        # 重载 "%" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] % n
    def _pow_(self,n):                        # 重载 "**" 运算符
        for i in range(0,len(self._mylist)):
            self._mylist[i] = self._mylist[i] ** n
    def _len_(self):                          # 重载 len 方法
        return len(self._mylist)
    def show(self):                           # 定义 show 方法
        print self._mylist

```

然后编写一个 UseMyList.py 脚本,使用 MyList.py 中的 MyList 类。其代码如下所示。

第6章 面向对象的Python

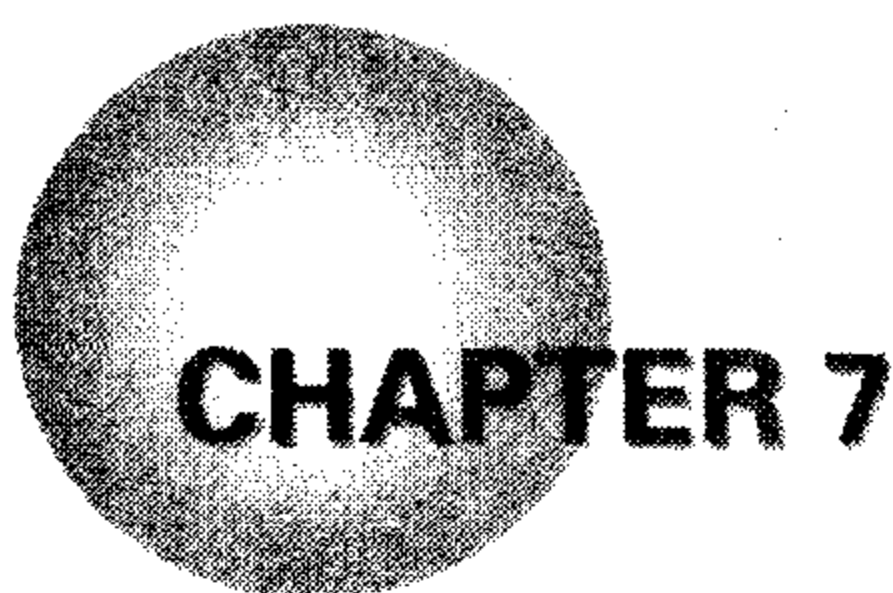
```
# -*- coding:utf-8 -*-
# file: UseMyList.py
#
import MyList
l = MyList.MyList(1,2,3,4,5)
l.show()
l + 10
l.show()
l * 2
l.show()
print len(l)
l ** 3
l.show()
```

运行 UseMyList.py 输出如下所示。

```
[1, 2, 3, 4, 5]
[11, 12, 13, 14, 15]
[22, 24, 26, 28, 30]
5
[10648, 13824, 17576, 21952, 27000]
```

```
# 导入 MyList 模块
# 调用 MyList 类实例化生成 l 对象
# 调用 show 方法
# 此处将调用 _add_ 方法
# 调用 show 方法
# 此处将调用 _mul_ 方法
# 调用 show 方法

# 此处将调用 _pow_ 方法
# 调用 show 方法
```



第 7 章 异常与调试

异常通常是脚本在运行过程中引发的错误。如果在脚本中未包含有关异常处理的代码，那么脚本将终止运行。在 Python 中可以为脚本添加异常处理，以应对可能出现的错误，从而使脚本更“健壮”。

7.1 捕获异常

在脚本运行过程中常见的异常有除零、下标越界等。在 Python 中可以捕获这些异常，并编写相关异常的处理语句。

7.1.1 使用 try 语句

在 Python 中可以使用 try 语句来处理异常。和 Python 中其他语句一样，try 语句也要使用缩进结构。try 语句也有一个可选的 else 语句块。一般的 try 语句形式如下所示。

```
try:
    <语句>                                # 要进行捕捉异常的语句
except <异常名 1>:
    <语句>                                # 要处理的异常
except <异常名 2>:
    <语句>                                # 对异常进行处理的语句
else:
    <语句>                                # 要进行处理的异常
    <语句>                                # 对异常进行处理的语句
    <语句>                                # 如果异常未触发，则执行该语句
```

try 语句在脚本中的执行过程如图 7-1 所示。

try 语句还有一种不包含 except 和 else 语句的特殊形式。其形式如下所示。

```
try:
    <语句>
finally:
    <语句>
```

不管 try 语句块中是否发生异常，都将执行 finally 语句块。以下实例使用 try 语句处理异常。

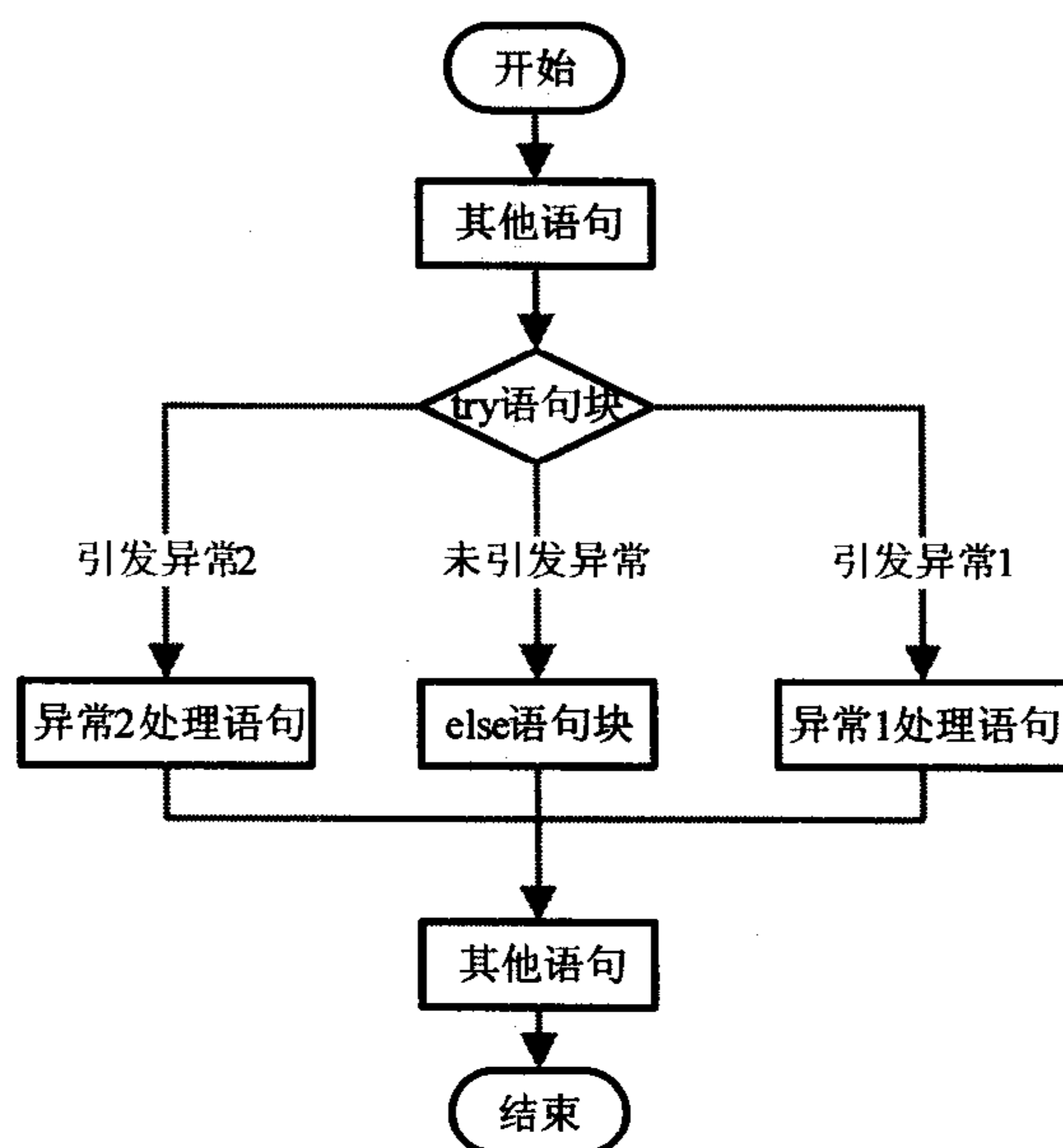


图 7-1 try 语句执行流程

```

>>> l = [1,2,3]                                # 定义一个列表
>>> l[5]                                         # 此处列表越界导致错误
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: list index out of range
>>> try:                                         # 使用 try 语句捕获异常
... l[5]                                         # 引发列表越界错误
... except:                                     # 此处未使用异常名, 表示捕获所有异常
...     print 'Error'                           # 如果引发异常, 则打印 "Error"
... else:                                       # 如果没有异常, 则将执行 else 语句块中的内容
...     print 'No Error'
...
Error                                           # 输出 Error 表示引发了异常
>>> try:
... l[2]                                         # 此处列表没有越界, 将不会引发异常
... except:
...     print 'Error'
... else:
...     print 'No Error'
...
3
No Error                                       # 此处为 else 语句块中的内容, 表示未引发异常
>>> try:
...     l[2]/0
... except IndexError:                          # 只捕获 IndexError 异常, 也就是列表越界异常
...     print 'Error'

```



```
... else:
...     print 'No Error'
...
# 脚本运行出错，因为在脚本中只捕获 IndexError 而没有捕获 ZeroDivisionError，也就是除零异常
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:                                     # 捕获多个异常，并分别处理
...     l[2]/0
... except IndexError:                         # 捕获 IndexError 异常
...     print 'IndexError'
... except ZeroDivisionError:                 # 捕获 ZeroDivisionError 异常
...     print 'ZeroDivisionError'
... else:
...     print 'No Error'
...
ZeroDivisionError
>>> try:                                     # 使用 try...finally 语句
...     l[2]                                  # 此处不会引发异常
... finally:
...     print 'A'                             # 仅打印字母“A”表示语句执行
...
3
A
>>> try:
...     l[5]                                  # 此处引发异常
... finally:
...     print 'A'
...
A                                             # 虽然引发异常，但 finally 语句块仍被执行
Traceback (most recent call last):
  File "<interactive input>", line 2, in <module>
IndexError: list index out of range
```

7.1.2 处理异常

在 except 中可以捕获指定的异常。除此以外，except 语句还可以捕获异常的附加数据。Python 中常用的内置异常如表 7-1 所示。

表 7-1 常用异常名

异常名	描述
AttributeError	调用不存在的方法引发的异常
EOFError	遇到文件末尾引发的异常
ImportError	导入模块出错引发的异常
IndexError	列表越界引发的异常
IOError	I/O 操作引发的异常，如打开文件出错等

续表

异常名	描述
KeyError	使用字典中不存在的关键字引发的异常
NameError	使用不存在的变量名引发的异常
TabError	语句块缩进不正确引发的异常
ValueError	搜索列表中不存在的值引发的异常
ZeroDivisionError	除数为零引发的异常

except 语句主要有以下几种用法。

```
except:                                # 捕获所有异常
except <异常名>:                        # 捕获指定异常
except (异常名 1, 异常名 2):           # 捕获异常名 1 或者异常名 2
except <异常名>, <数据>:                # 捕获指定异常及其附加的数据
except (异常名 1, 异常名 2), <数据>:   # 捕获异常名 1 或者异常名 2 及异常的附加数据
以下实例使用 except 捕获异常。
>>> l = [1,2,3]                        # 定义一个列表
>>> try:                                # 异常处理
...   l[5]
... except IndexError, Error:           # 捕获 IndexError 异常并获取其附加数据
...   print Error                      # 打印异常的附加数据
... else:
...   print 'No Error'                 # 如果未触发异常，则打印 “No Error”
...
list index out of range                # 此处为异常的附加数据
>>> try:
...   l[2]/0
... except (IndexError, ZeroDivisionError): # 捕获 IndexError 异常或者
                                           ZeroDivisionError 异常
...   print 'Error'
... else:
...   print 'No Error'
...
Error
>>> try:
...   l[5]/0
... except:                             # 捕获所有异常
...   print 'Error'
... else:
...   print 'No Error'
...
Error
>>> try:
...   l[2]/0
... except (IndexError, ZeroDivisionError), value: # 捕获 IndexError 异常或者
                                                    ZeroDivisionError 异常及其附加数据
...   print value
```

```
... else:
...     print 'No Error'
...
integer division or modulo by zero
```

7.1.3 多重异常处理

在 Python 中可以在 try 语句中嵌套另外一个 try 语句。由于 Python 将 try 语句放在堆栈中，一旦引发异常，Python 将匹配最近的 except 语句。如果 except 能够处理该异常，则外围的 try 语句将不会捕获异常。如果 except 忽略该异常，则异常将被外围 try 语句捕获。如下所示代码在 try 语句中嵌套另外一个 try 语句。

```
>>> l = [1,2]                # 定义一个列表
>>> try:                      # 嵌套 try 语句
...     try:
...         l[5]
...     except:               # 捕获所有异常
...         print 'Error1'    # 打印 "Error1"
...     except:               # 捕获所有异常
...         print 'Error2'    # 打印 "Error2"
...     else:
...         print 'No Error'
...
Error1                        # 此处为内嵌 try 语句输出
No Error                     # 此处为外围 try 语句输出，表示 try 语句块中未引发异常
>>> try:
...     try:
...         l[1]/0
...     except IndexError:    # 仅捕获 IndexError 异常
...         print 'Error1'
...     except:               # 捕获所有异常
...         print 'Error2'
...     else:
...         print 'No Error'
...
Error2                        # 异常被外围 try 语句捕获
>>> try:
...     try:
...         l[1]/'s'
...     except IndexError:    # 仅捕获 IndexError 异常
...         print 'Error1'
...     except ZeroDivisionError: # 仅捕获 ZeroDivisionError 异常
...         print 'Error2'
...     else:
...         print 'No Error'
...
Traceback (most recent call last): # 最终脚本执行出错
  File "<interactive input>", line 3, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

7.2 引发异常

除了内置的异常以外，在 Python 中还可以通过使用 `raise` 语句手工引发异常。在类中也可以使用 `raise` 引发异常，并向异常传递数据。使用 `raise` 可以定义新的错误类型，以适应脚本的需要。例如对用户输入数据的长度有要求，则可以使用 `raise` 引发异常，以确保数据输入符合要求。

7.2.1 使用 `raise` 引发异常

使用 `raise` 引发异常十分简单。`raise` 有以下几种使用方式。

```
raise 异常名
raise 异常名, 附加数据
raise 类名
```

以下实例使用 `try` 语句捕获由 `raise` 引发的异常。

```
>>> raise 'Exception' # 使用 raise 引发异常
_main_:1: DeprecationWarning: raising a string exception is deprecated
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
Exception
>>> try: # 使用 try 语句捕获 raise 引发的异常
...     raise 'Exception'
... except 'Exception': # 捕获 "Exception"
...     print 'Error'
... else:
...     print 'No Error'
...
Error
>>> try:
...     raise 'Exception', 'Raise an Exception' # 使用 raise 引发异常并传递附加数据
... except 'Exception', data: # 捕获异常和附加数据
...     print data
... else:
...     print 'No Error'
...
Raise an Exception
>>> def fun(n): # 定义一个函数
...     if n == 0:
...         raise 'Zero', 'n is zero' # 在函数中使用 raise 引发异常
...     else:
...         print n
...
>>> try: # 使用 try 语句捕获函数中的异常
...     fun(0)
... except 'Zero', data:
...     print data
```



```
...
_main_:3: DeprecationWarning: raising a string exception is deprecated
n is zero
>>> class A:                                # 定义一个类
...     def show(self):                      # 定义类的 show 方法
...         print 'A'
...
>>> try:
...     raise A                                # 使用 raise 引发一个类异常
... except A:
...     print 'Error'
... else:
...     print 'No Error'
...
Error
```

7.2.2 assert——简化的 raise 语句

在 Python 中使用 assert 语句同样可以引发异常。但与 raise 语句不同，assert 语句是在条件测试为假时，才引发异常。assert 语句的一般形式如下所示。

```
assert <条件测试>, <异常附加数据>           # 其中异常附加数据是可选的
```

以下实例使用 assert 语句引发异常。

```
>>> l = []                                   # 定义一个列表
>>> assert len(l)                           # 如果列表为空，则使用 assert 引发异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AssertionError
>>> assert len(l), 'Error'                  # 向异常传递附加数据
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AssertionError: Error
>>> try :                                  # 使用 try 语句捕获 assert 异常
...     assert len(l), 'Error'
... except:                                # 捕获所有异常
...     print 'Error'
... else:
...     print 'No Error'
...
Error
>>> l.append(1)                             # 向列表中添加成员
>>> assert len(l)                           # 此时列表不为空，assert 将不会引发异常。
```

从上述实例可以看出，assert 相当于 raise 语句和 if 语句联合使用。例如如下 assert 语句。

```
assert len(l)
```

可以改写为如下 raise 语句。

```
if _debug_:
    if len(l):
        raise AssertionError, <附加数据>
```

需要注意的是，`assert` 语句一般用于开发时对程序条件的验证。只有当内置 `_debug_` 为 `True` 时，`assert` 语句才有效。当 Python 脚本以 `-O` 选项编译成为字节码文件时，`assert` 语句将被移除。

7.2.3 自定义异常类

在 Python 中可以通过继承 `Exception` 类来创建自己的异常类。异常类和其他的类并没有区别，一般仅在异常类中定义几个属性信息。以下实例通过继承 `Exception` 类来生成一个 `MyError` 类。

```
>>> class MyError(Exception):           # 通过继承 Exception 类来生成 MyError 类
...     def __init__(self,data):         # 重载 __init__ 方法
...         self.data = data
...     def __str__(self):               # 重载 __str__ 方法
...         return self.data
...
>>> raise MyError,'Error'               # 使用 raise 引发 MyError 异常
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
MyError: Error
>>> try:                                # 使用 try 语句捕获 MyError 异常
...     raise MyError,'Raise MyError'
... except MyError,data:                 # 捕获异常和附加数据
...     print data
... else:
...     print 'No error'
...
Raise MyError
```

7.3 使用 pdb 调试 Python 脚本

在 Python 中脚本的语法错误可以被 Python 解释器发现，但是脚本逻辑上的错误，或者其他的一些变量使用错误却不容易被发现。如果脚本运行后没有获得预想的结果，则需要对脚本进行调试。`pdb` 模块是 Python 自带的调试模块。使用该模块可以为脚本设置断点、单步执行、查看变量值等。`pdb` 模块可以以命令行参数的形式启动，也可以通过 `import` 将其导入使用。

通过 `import` 导入 `pdb` 模块后，就可以使用 `pdb` 模块的函数对脚本进行调试。常用的 `pdb` 模块的函数可以分为以下几类。

7.3.1 运行语句

在 Python 中可以使用 `pdb` 模块的 `run` 函数来调试语句块。其参数原型如下所示。

```
run( statement[, globals[, locals]])
```

其参数含义如下。

- **statement**: 要调试的语句块, 以字符串的形式。
- **globals**: 可选参数, 设置 **statement** 运行的全局环境变量。
- **locals**: 可选参数, 设置 **statement** 运行的局部环境变量。

以下实例使用 **run** 函数调试语句块。

```
>>> import pdb                                # 导入 pdb 模块
>>> pdb.run('''
... for i in range(0,3):
...     i = i ** 2
...     print i
... ''')
> <string>(2)<module>()->None                  # “(Pdb)” 为调试命令提示符, 表示可以输入调试命令
(Pdb) n                                        # 执行下一行
> <string>(3)<module>()->None
(Pdb) n
> <string>(4)<module>()->None
(Pdb) print i                                # print 打印变量 i 的值
0
(Pdb) continue                               # 继续运行程序
0
1
4
```

7.3.2 运行表达式

在 Python 中可以使用 **pdb** 模块的 **runeval** 函数来调试表达式。其参数原型如下所示。

```
runeval( expression[, globals[, locals]])
```

其参数含义如下。

- **expression**: 要调试的表达式, 以字符串的形式。
- **globals**: 可选参数, 设置 **statement** 运行的全局环境变量。
- **locals**: 可选参数, 设置 **statement** 运行的局部环境变量。

以下实例使用 **runeval** 函数来调试表达式。

```
>>> import pdb                                # 导入 pdb 模块
>>> l = [1,2,3]                               # 定义一个列表
>>> pdb.runeval('l[1]')                       # 使用 runeval 调试表达式 l[1]
> <string>(1)<module>()->2
(Pdb) n                                        # 进入调试状态, 使用 n 命令, 单步执行
--Return--
> <string>(1)<module>()->2
(Pdb) n                                        # 使用 n 命令, 单步执行
2                                              # 表达式的值
>>> pdb.runeval('3+5*6/2')                   # 使用 runeval 调试表达式 3+5*6/2
> <string>(1)<module>()->2
(Pdb) n                                        # 进入调试状态, 使用 n 命令, 单步执行
--Return--
```

```
> <string>(1)<module>()->18
(Pdb) n                                # 使用 n 命令，单步执行
18                                     # 表达式的值
```

7.3.3 运行函数

在 Python 中可以使用 pdb 模块的 runcall 函数来调试函数。其函数原型如下所示。

```
runcall( function[, argument, ...])
```

其参数含义如下。

- function: 函数名。
- argument: 函数的参数。

以下实例使用 runcall 函数来调试函数。

```
>>> import pdb                        # 导入 pdb 模块
>>> def sum(*args):                  # 定义函数 sum 求所有参数之和
...     r = 0
...     for arg in args:
...         r = r + arg
...     return r
...
>>> pdb.runcall(sum,1,2,3,4)         # 使用 runcall 调试函数 sum
> <stdin>(2)sum()
(Pdb) n                              # 进入调试状态，使用 n 命令，单步执行
> <stdin>(3)sum()
(Pdb) n                              # 使用 n 命令，单步执行
> <stdin>(4)sum()
(Pdb) print r                        # 使用 print 打印变量 r 的值
0
(Pdb) continue                      # 使用 continue 继续执行
10                                  # 函数返回值
>>> pdb.runcall(sum,1,2,3,4,5,6)    # 使用 runcall 调试函数 sum
> <stdin>(2)sum()
(Pdb) continue                      # 使用 continue 继续执行
21                                  # 函数返回值
```

7.3.4 设置硬断点

在 Python 中可以使用 pdb 模块的 set_trace 函数在脚本中设置硬断点。set_trace 函数一般在“.py”脚本中使用。其函数原型如下所示。

```
set_trace()
```

以下实例使用 run 函数调试语句块。

```
# -*- coding:utf-8 -*-
# file: debug.py
#
import pdb                            # 导入 pdb 模块
pdb.set_trace()                      # 使用 set_trace 函数设置硬断点
for i in range(0,5):
```



```
i = i * 5
print i

运行脚本后如下所示。

> e:\book\code\debug.py(6)<module>()
-> for i in range(0,5):
(Pdb) list                                     # 使用 list 列出脚本内容
1      # -*- coding:utf-8 -*-
2      # file: debug.py
3      #
4      import pdb
5      pdb.set_trace()                         # 使用 set_trace 函数设置硬断点
6  -> for i in range(0,5):
7          i = i * 5
8          print i
[EOF]
(Pdb) continue                                # 使用 continue 继续执行
0
5
10
15
20
```

7.3.5 pdb 调试命令

在上一节中已经使用了部分 pdb 的调试命令与被调试的脚本进行交互。pdb 中的调试命令可以完成单步执行、打印变量值、设置断点等功能。pdb 的部分调试命令如表 7-2 所示。

表 7-2		pdb 调试命令	
完整命令	简写命令	描 述	
args	a	打印当前函数的参数	
break	b	设置断点	
clear	cl	清除断点	
condition	无	设置条件断点	
continue	c 或者 cont	继续运行，直到遇到断点或者脚本结束	
disable	无	禁用断点	
enable	无	启用断点	
help	h	查看 pdb 帮助	
ignore	无	忽略断点	
jump	j	跳转到指定行数运行	
list	l	列出脚本清单	
next	n	执行下条语句，遇到函数不进入其内部	

续表

完整命令	简写命令	描述
p	p	打印变量值，也可以用 print
quit	q	退出 pdb
return	r	一直运行到函数返回
tbreak	无	设置临时断点，断点只中断一次
step	s	执行下一条语句，遇到函数进入其内部
where	w	查看所在的位置
!	无	在 pdb 中执行语句

以下实例通过命令行启动 pdb 对脚本进行调试。在 Windows 的命令行中进入脚本所在的目录，输入以下命令启动 Python。

```
E:\book\code>python -m pdb prime.py
```

脚本运行后输入以下命令。

```
> e:\book\code\prime.py(4)<module>()
-> import math
(Pdb) list
```

先停在这里，前边有“(Pdb)”提示符，
输入 list 命令
list 命令默认只列出前 11 行

```
1      # -*- coding:utf-8 -*-
2      # file: prime.py
3      #
4      -> import math
5      # isprime 函数判断一个整数是否为素数。
6      # 如果 i 能被 2 到 i 的平方根内的任意一个数整除，
7      # 则 i 不是素数，返回 0，否则 i 是素数，返回 1。
8      def isprime(i):
9          for t in range( 2, int(math.sqrt(i)) + 1 ):
10             if i % t == 0:
11                 return 0
```

```
(Pdb) l 14,17
```

使用 list 命令列出 14 到 17 行的脚本内容

```
14      print '100~110 之间的素数有： '
15      for i in range(100,110):
16          if isprime(i):
17              print i
```

```
(Pdb) b 14
```

使用 b 在第 14 行设置断点
返回断点编号 1

```
Breakpoint 1 at e:\book\code\prime.py:14
```

```
(Pdb) b isprime
```

在函数 isprime 上设置断点
返回断点编号 2

```
Breakpoint 2 at e:\book\code\prime.py:8
```

```
(Pdb) c
```

使用 c 命令运行脚本

```
> e:\book\code\prime.py(14)<module>()
```

停在断点 1 处，即第 14 行

```
-> print '100~110 之间的素数有： '
```

```
(Pdb) c
```

使用 c 命令继续运行脚本
第 14 行脚本输出

```
100~200 之间的素数有：
```

```

> e:\book\code\prime.py(9)isprime()          # 停在断点 2 处, 即 isprime 函数处
-> for t in range( 2, int(math.sqrt(i)) + 1 ):
(Pdb) b 15                                     # 在第 15 行设置断点
Breakpoint 3 at e:\book\code\prime.py:15      # 返回断点编号 3
(Pdb) disable 2                               # 禁用断点 2, 即 isprime 函数处的断点
(Pdb) c                                       # 继续运行脚本
> e:\book\code\prime.py(15)<module>()         # 停在断点 3 处, 即第 15 行
-> for i in range(100,110):
(Pdb) print i                                 # 使用 print 打印变量 i 的值
100
(Pdb) c                                       # 继续运行脚本
101
> e:\book\code\prime.py(15)<module>()         # 停在断点 3 处, 即第 15 行
-> for i in range(100,110):
(Pdb) p i                                     # 使用 p 打印变量 i 的值
101
(Pdb) enable 2                               # 恢复断点 2, 即 isprime 函数处的断点
(Pdb) c                                       # 继续运行脚本
> e:\book\code\prime.py(9)isprime()         # 停在断点 2 处, 即 isprime 函数处的断点
-> for t in range( 2, int(math.sqrt(i)) + 1 ):
(Pdb) n                                       # 单步执行下一语句
> e:\book\code\prime.py(10)isprime()
-> if i % t == 0:
(Pdb) print t                                # 停在下一语句处
2                                             # 使用 print 打印变量 t 的值
(Pdb) cl                                     # 清除所有断点, 输入 y 确认
Clear all breaks? y
(Pdb) c                                       # 继续运行脚本
103
105
107
109
The program finished and will be restarted   # 脚本运行结束, 回到开始处
> e:\book\code\prime.py(4)<module>()
-> import math
(Pdb) q                                       # 使用 q 命令退出 pdb

```










7.4 在 PythonWin 中调试 Python 脚本

在 PythonWin 中集成了 Python 脚本调试环境, 可以方便地对脚本进行调试。如果觉得使用 pdb 的调试命令过于繁琐的话, 可以使用 PythonWin 的脚本调试功能。运行 PythonWin 后单击【View】|【Toolbars】|【Debugging】命令, 将其前边打勾, 如图 7-2 所示。将会出现如图 7-3 所示的调试工具栏。

其中各按钮说明如下。

- : 查看变量或者表达式的值。

第7章 异常与调试

- : 查看堆栈。
- : 查看断点列表。
- : 设置断点。
- : 清除断点。
- : 单步执行, 相当于 step。
- : 单步执行, 相当于 next。
- : 执行到返回, 相当于 return。
- : 执行脚本。
- : 退出调试。

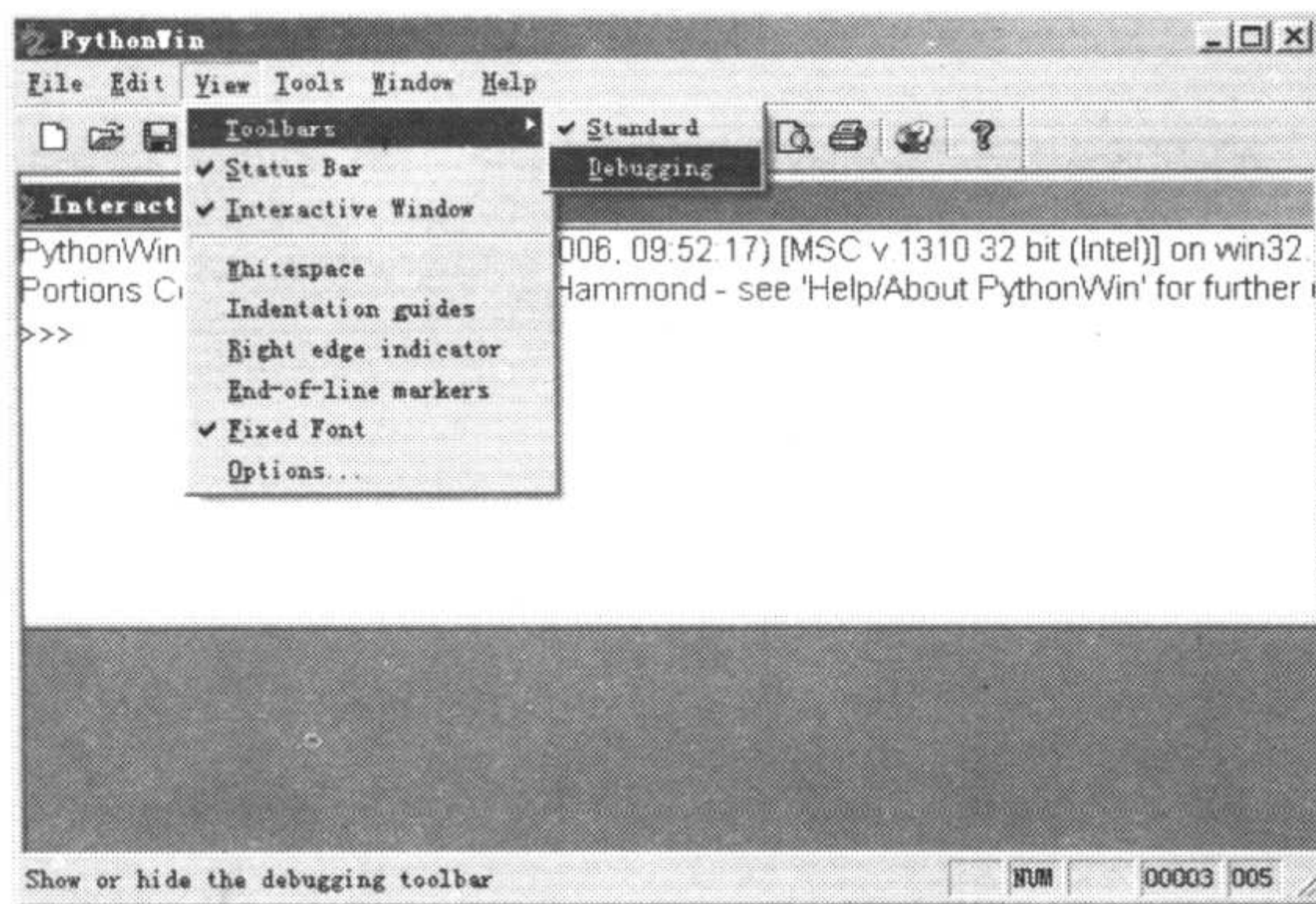


图 7-2 显示调试工具栏

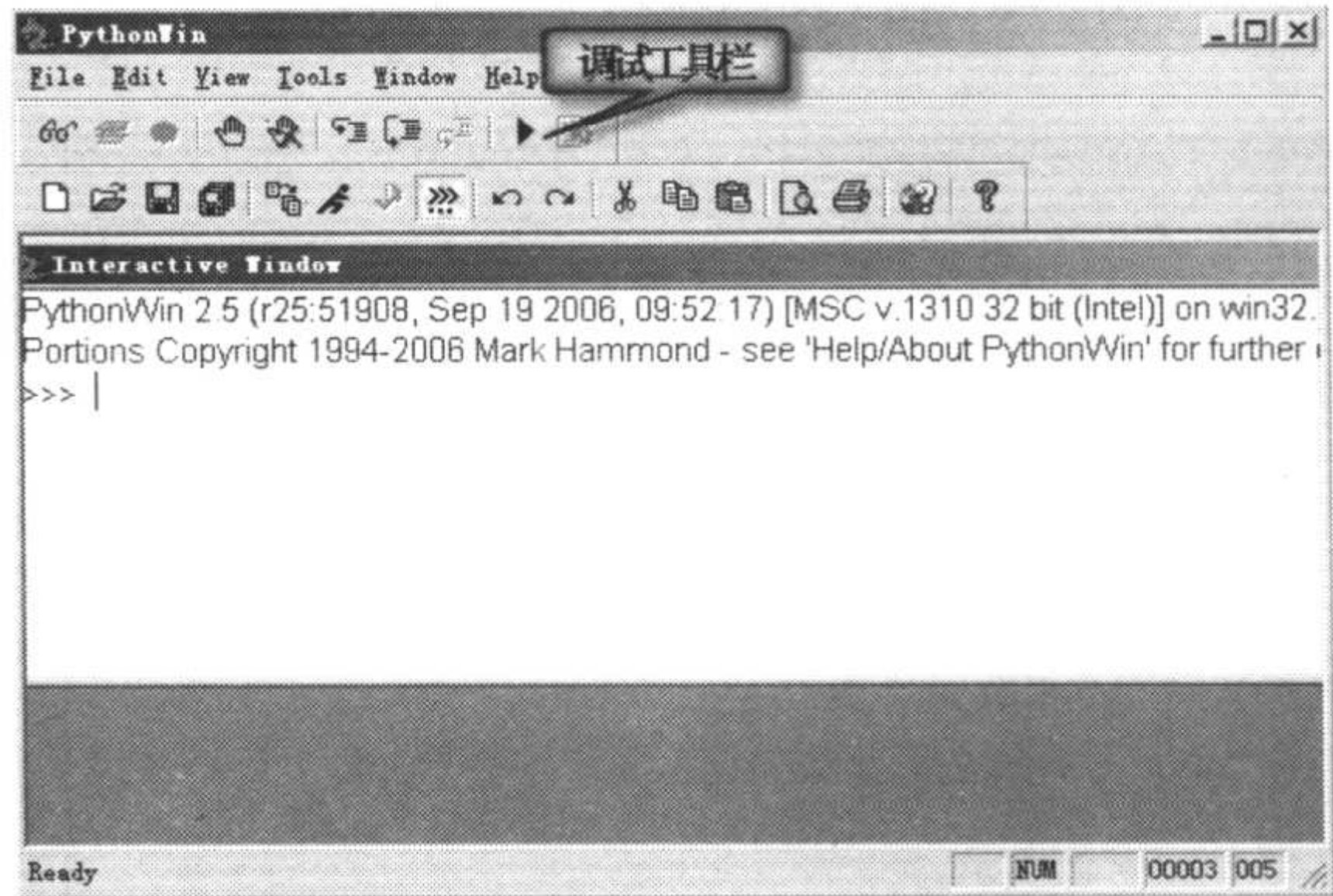



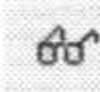





图 7-3 PythonWin 中的调试工具栏

以调试 prime.py 脚本为例, 在 PythonWin 中调试脚本的过程如下。

- (1) 打开 prime.py 脚本。
- (2) 将光标移动到第 9 行, 单击设置断点按钮  在第 9 行设置断点。
- (3) 将光标移动到第 14 行, 单击设置断点按钮  在第 14 行设置断点。断点设置完成后, 如图 7-4 所示。
- (4) 单击运行脚本按钮  执行脚本, 此时脚本中断在第 14 行, 如图 7-5 所示。
- (5) 单击查看按钮  , 将出现如图 7-6 所示的浮动窗口。
- (6) 双击浮动窗口中的 “New Item”, 添加查看 i 变量的值。
- (7) 单击运行脚本按钮  执行脚本, 脚本将中断在第 9 行, 如图 7-7 所示。可以看到浮动窗口中 i 的值为 100。
- (8) 单击显示断点列表按钮  , 将打开如图 7-8 所示的断点列表窗口。
- (9) 单击清除断点按钮  , 将清除所有断点, 如图 7-9 所示。

征服 Python——语言基础与典型应用

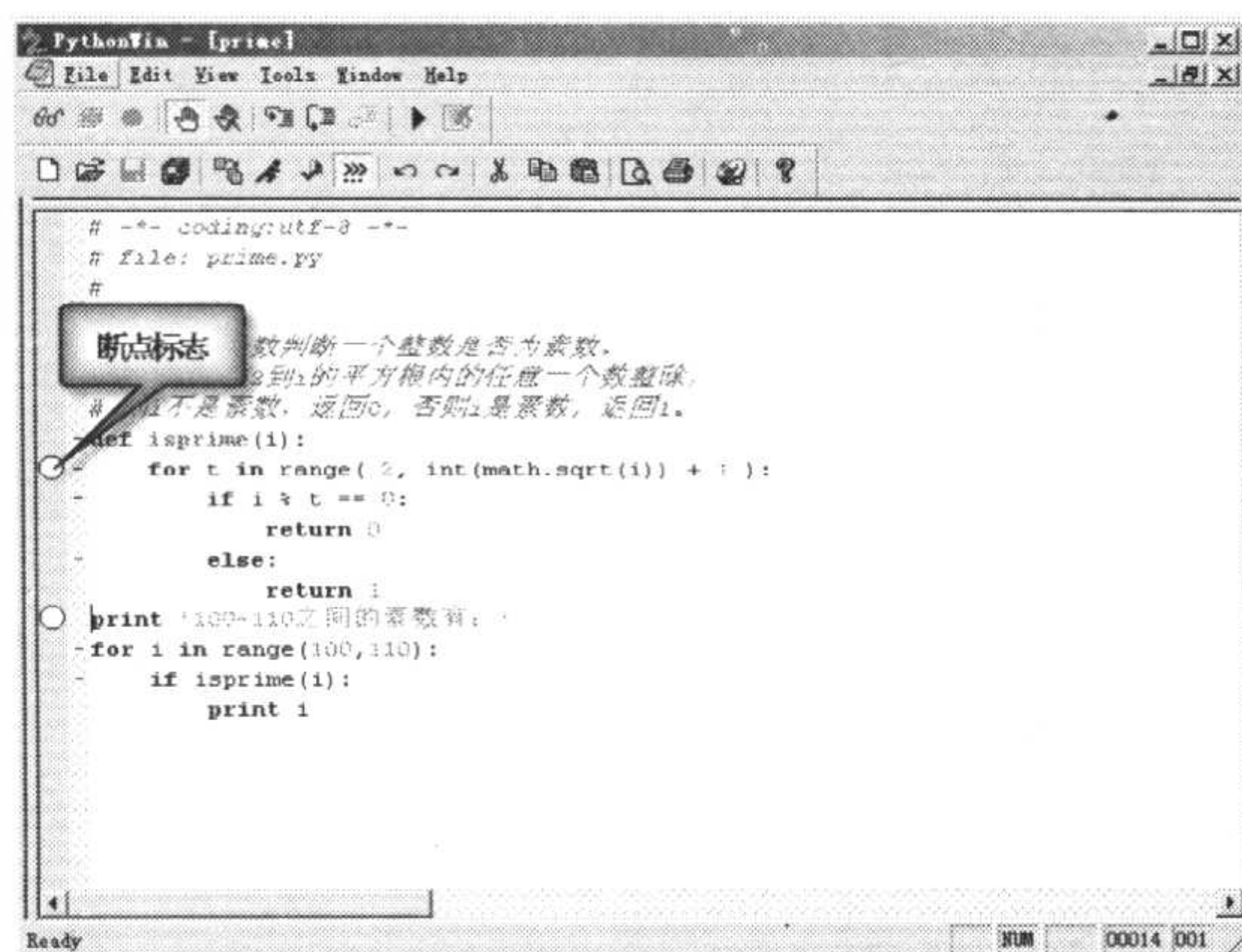


图 7-4 在 PythonWin 中设置断点

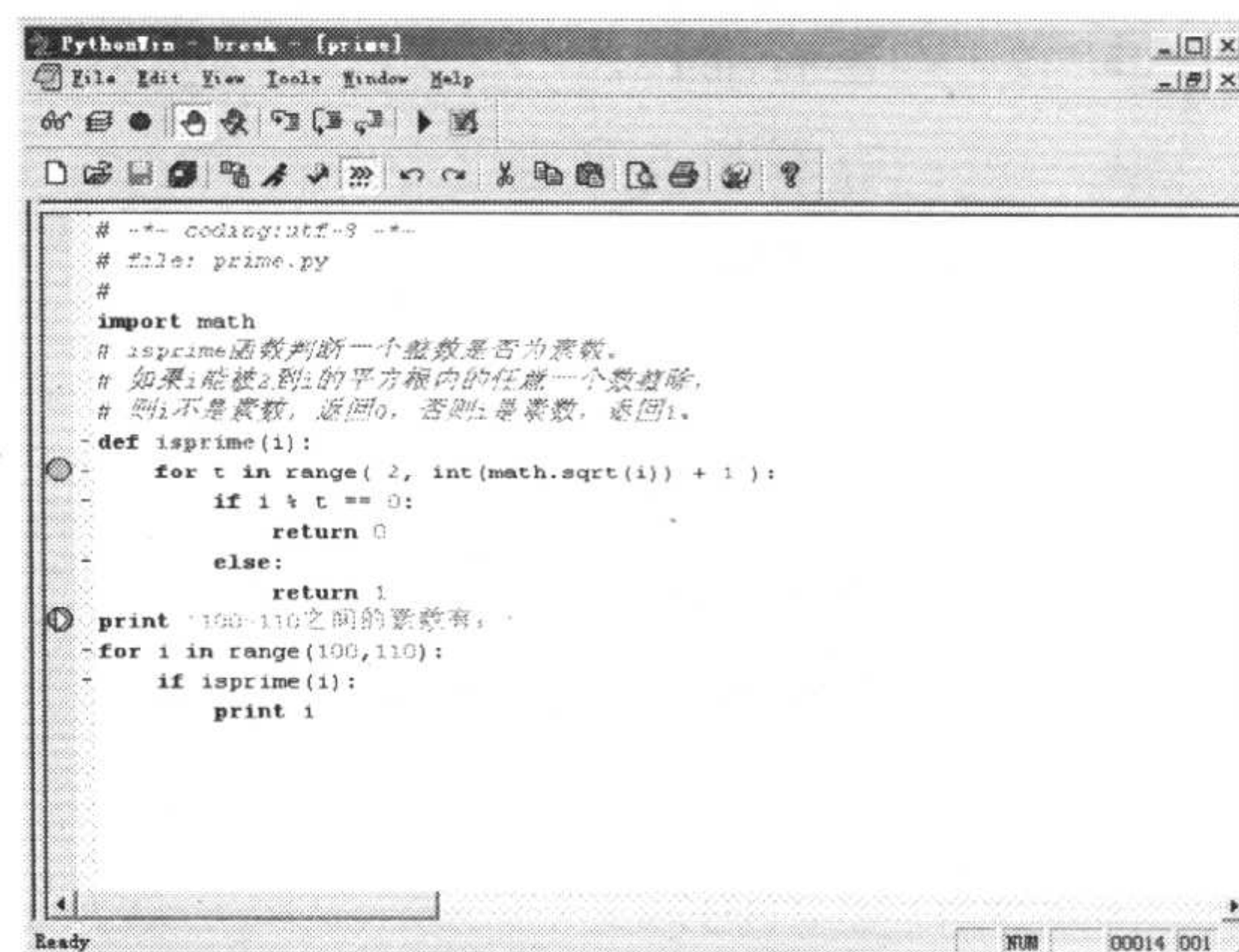


图 7-5 脚本中断在第 14 行

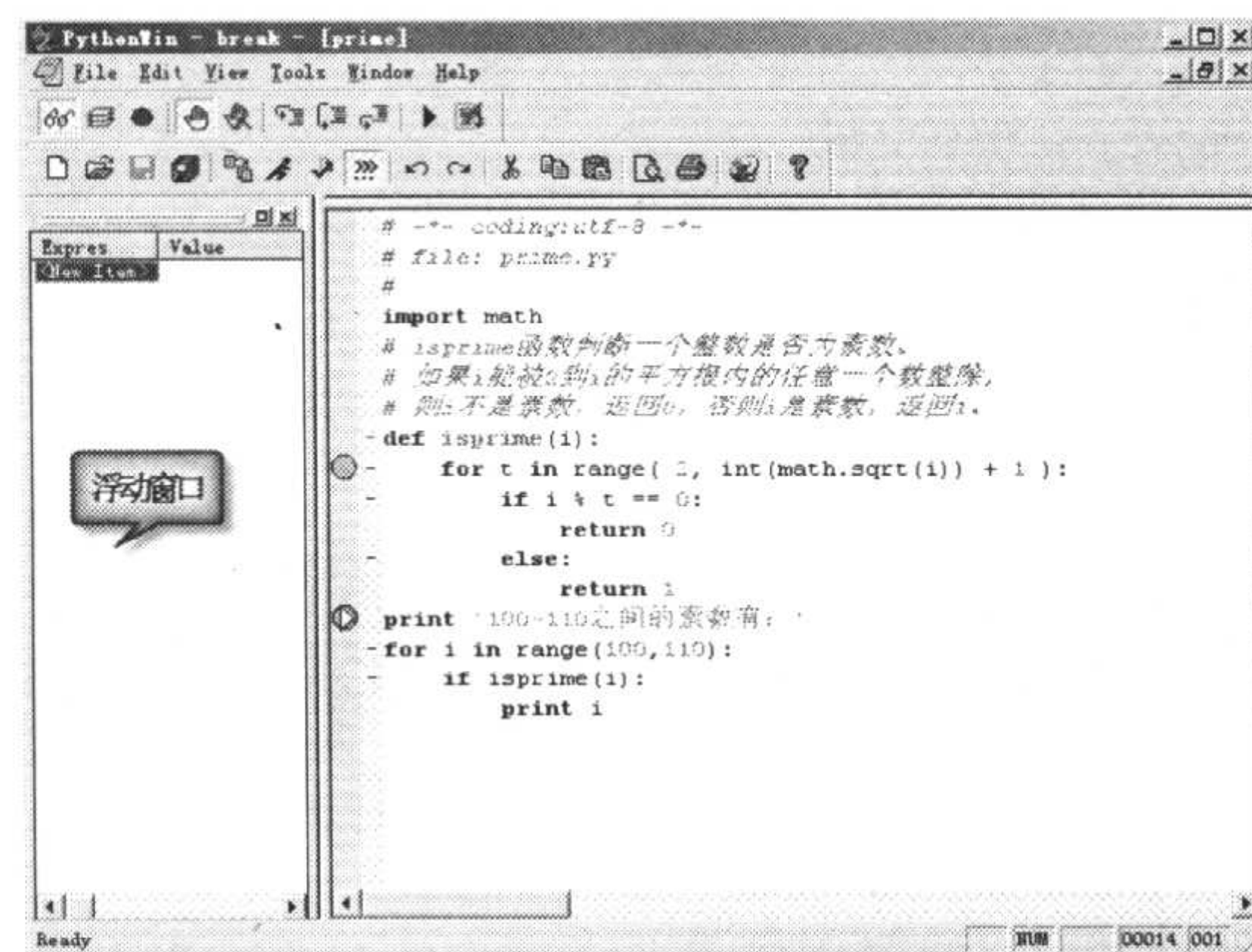


图 7-6 PythonWin 中的浮动窗口

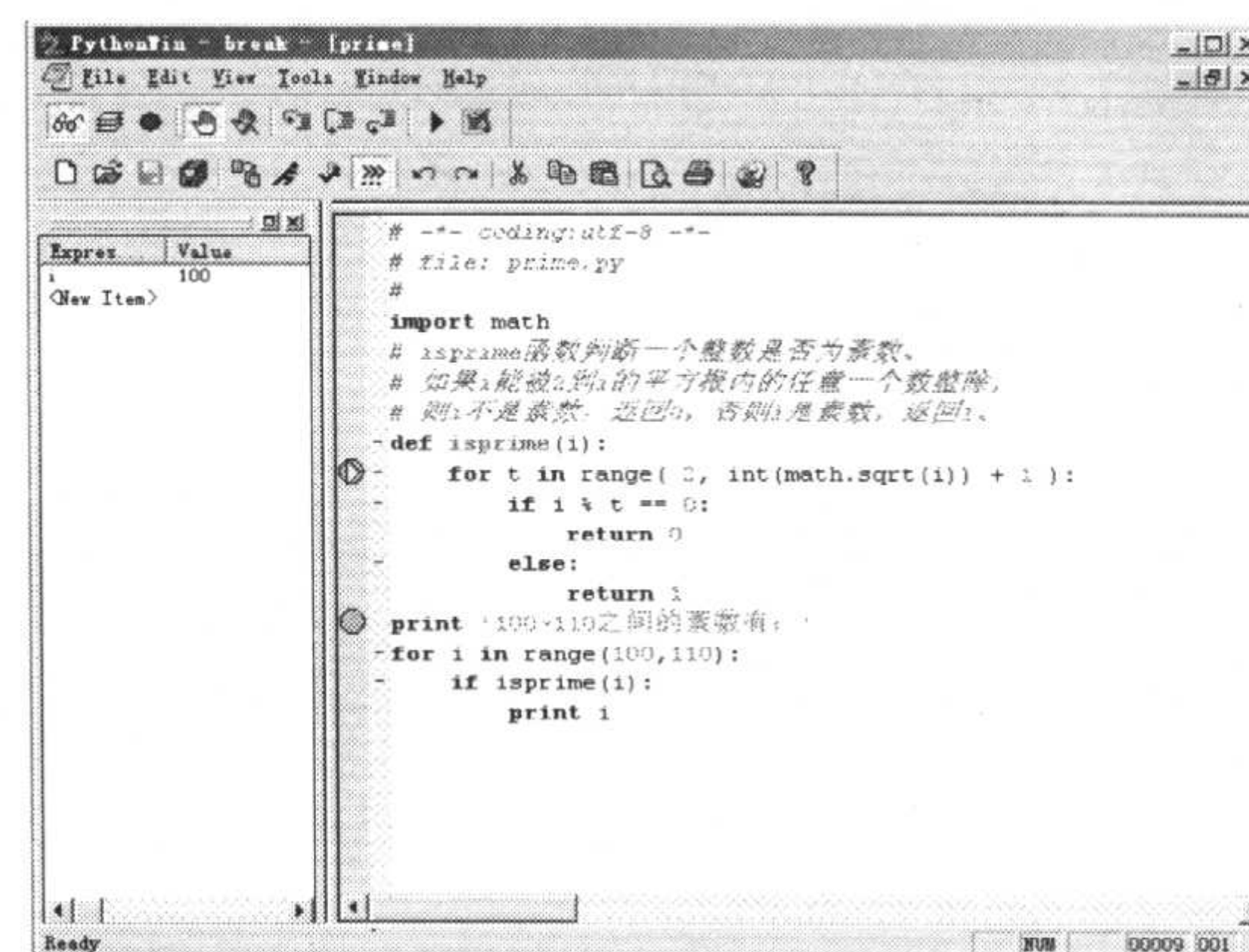


图 7-7 脚本中断在第 9 行

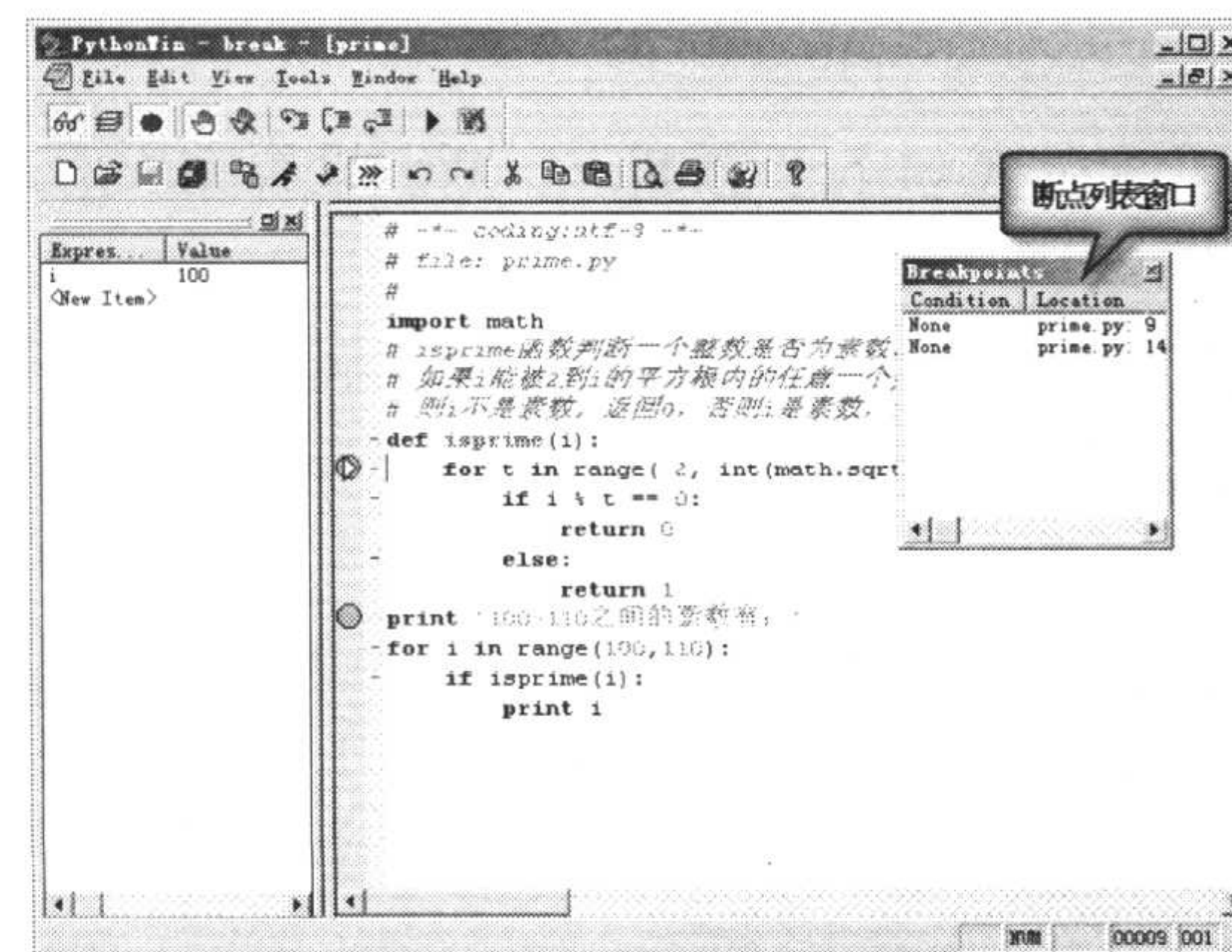


图 7-8 断点列表窗口

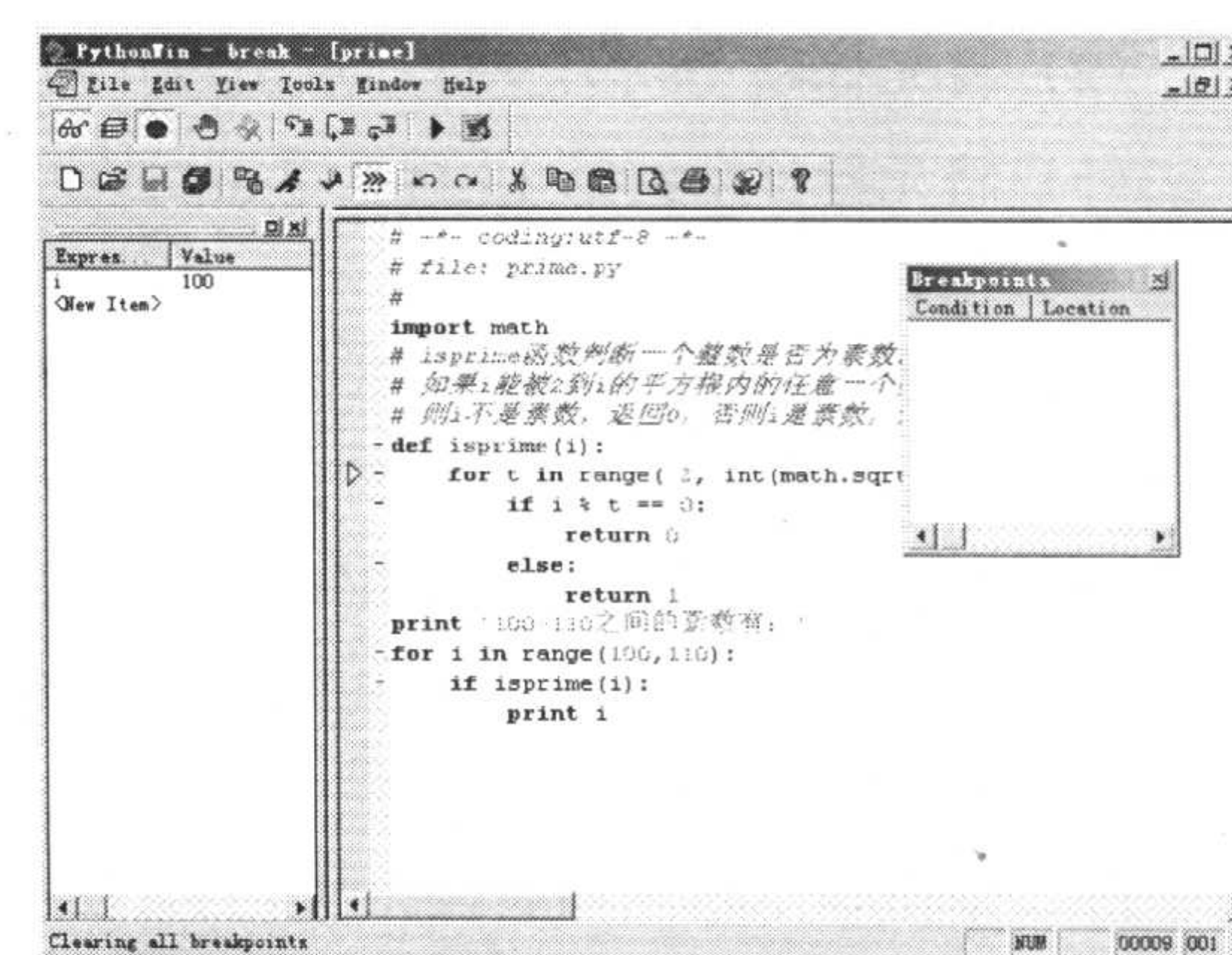



图 7-9 清除所有断点

第7章 异常与调试

(10) 单击退出调试按钮，将退出调试状态，如图 7-10 所示。

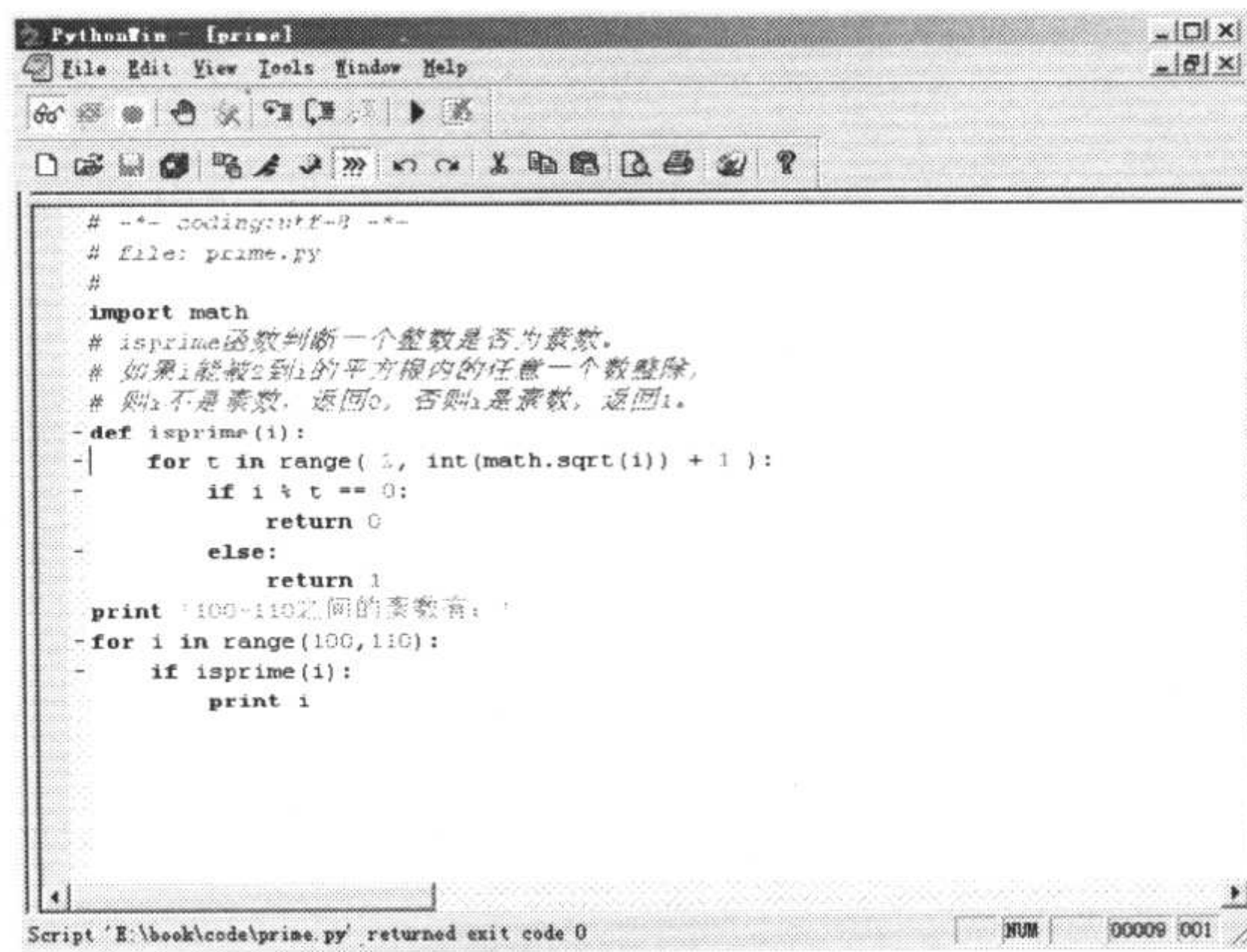
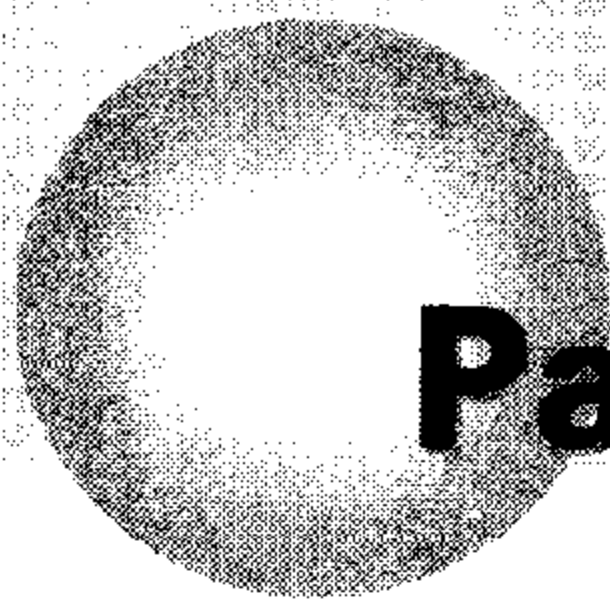


图 7-10 退出调试



Part 3

第三篇

系统应用

第 8 章 Python 扩展和嵌入

由于 Python 是解释性的脚本语言，执行速度较慢。在某些需要提高脚本执行效率的情况下，可以考虑扩展 Python，用 C/C++ 来完成对效率要求高的部分。另外，由于 Python 功能强大，完全可以将其嵌入到 C/C++ 中，以简化程序，减少代码。

8.1 扩展 Python

Python 提供了支持 C/C++ 接口，可以方便地使用 C/C++ 来扩展 Python。用 C/C++ 编写的 Python 扩展主要用于完成底层的系统操作，以及提高执行速度等。

8.1.1 扩展概述

Python 提供了接口 API，通过使用 API 函数可以编写 Python 扩展。在 Windows 下可以使用 VC 来编译 Python 扩展。在 UNIX 和 Linux 下则可以使用 gcc 来编译。

1. 设置编程环境

使用 VC 时需要设置一些头文件以及库文件的包含目录。如果使用 VC++ 6.0，则设置过程如下所示。

(1) 单击 **【Tools】|【Options】** 命令，弹出如图 8-1 所示的对话框。

(2) 单击 **【Directories】** 标签，选择 **【Show directories for】** 下拉列表框中的 **【Include files】** 项，将 Python 安装目录下的 INCLUDE 目录添加到 **【Directories】** 列表中，如图 8-2 所示。

(3) 选择 **【Show directories for】** 下拉列表框中的 **【Library files】** 项，将 Python 安装目录下的 LIBS 目录添加到 **【Directories】** 列表中，如图 8-3 所示。

(4) 单击 **【OK】** 按钮完成操作。

如果使用 Visual Studio 2005，则设置过程如下所示。

(1) 单击 **【工具】|【选项】** 命令，弹出如图 8-4 所示的对话框。

(2) 双击左侧列表树中的 **【项目和解决方案】** 项，选择 **【VC++ 目录】** 项，如图 8-5 所示。

(3) 选择 **【显示以下内容的目录】** 下拉列表框中的 **【包含文件】** 项，将 Python 安装目录

征服 Python——语言基础与典型应用

下的 include 目录添加到列表中，如图 8-6 所示。

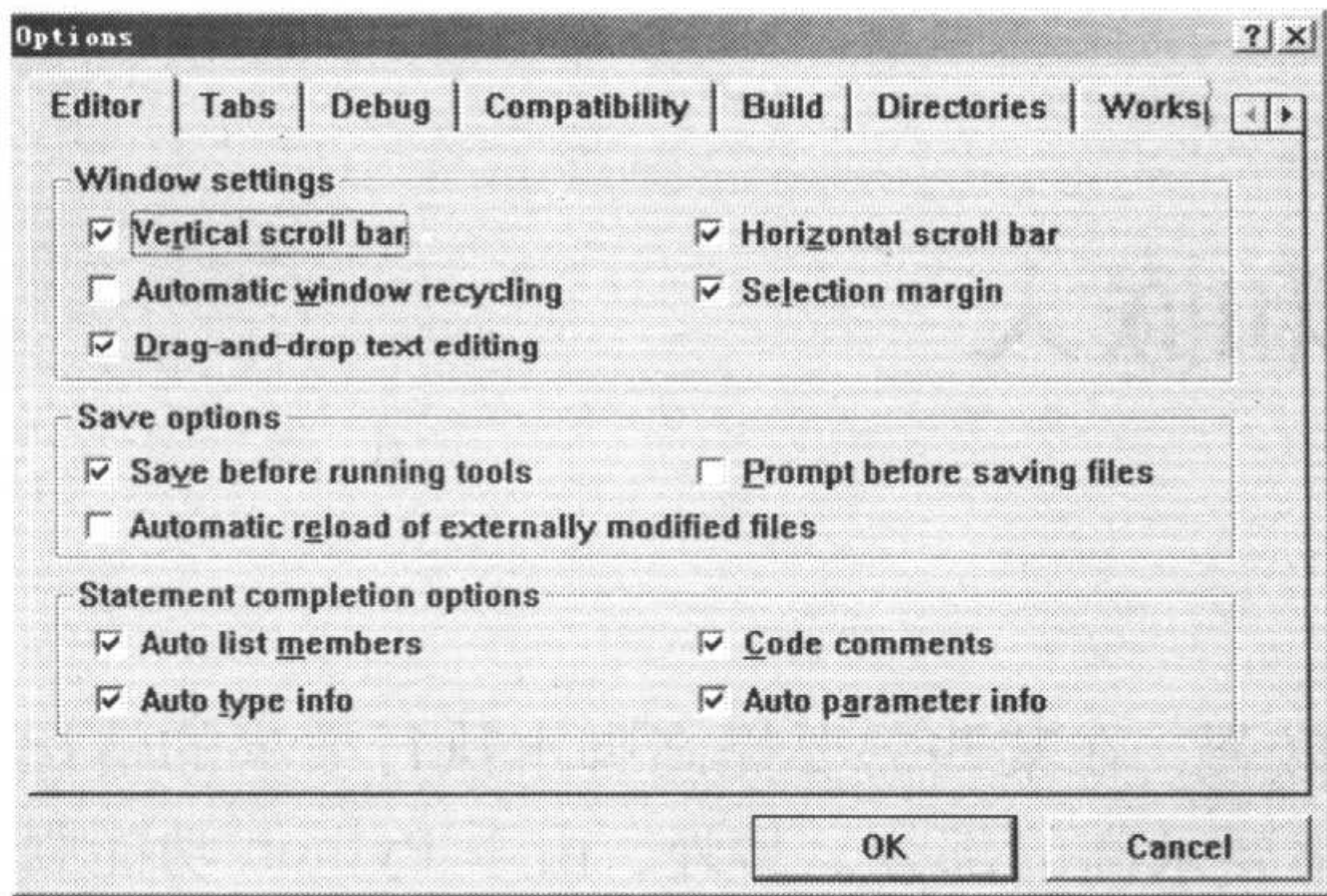


图 8-1 Options 对话框

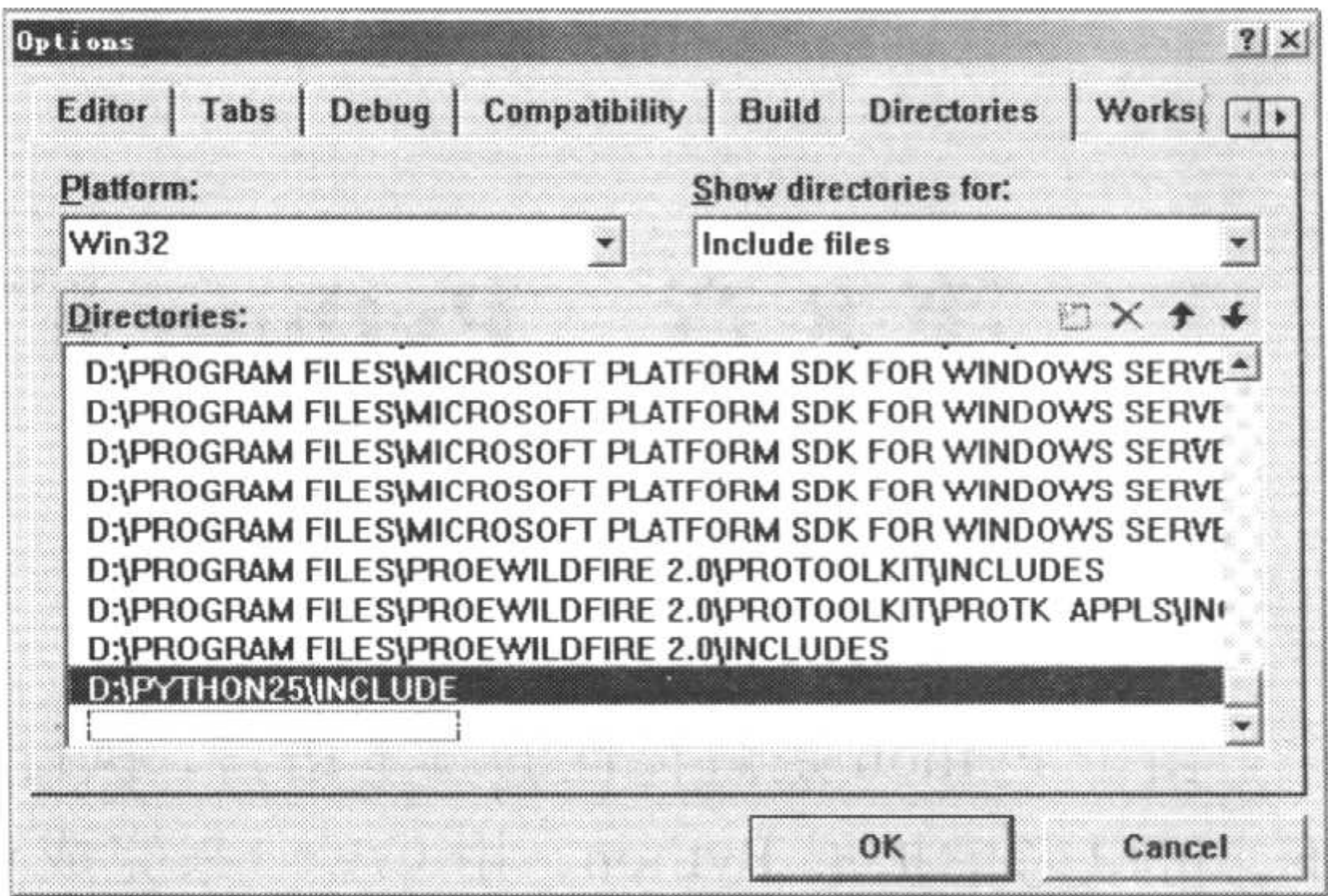


图 8-2 添加头文件

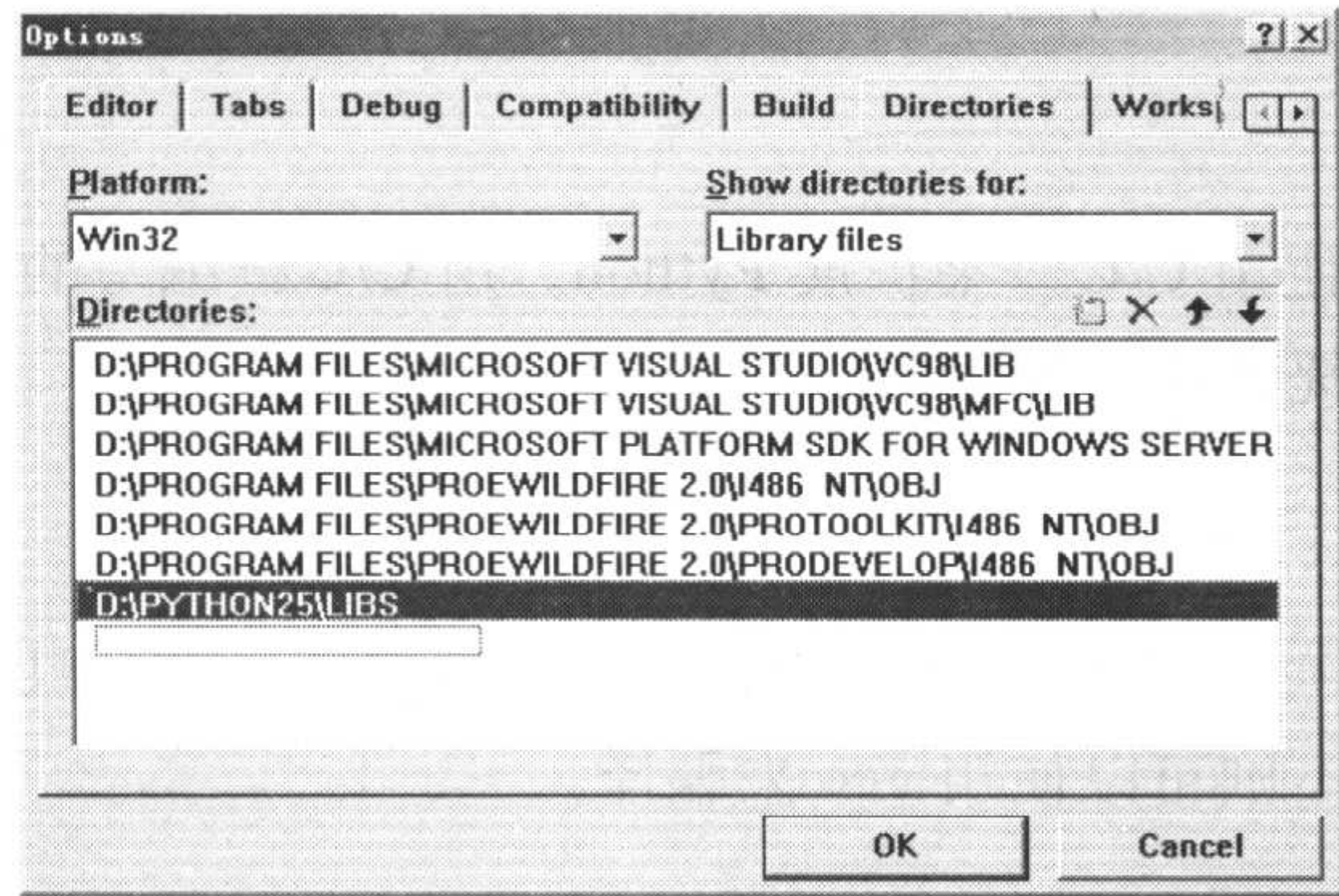


图 8-3 添加库文件

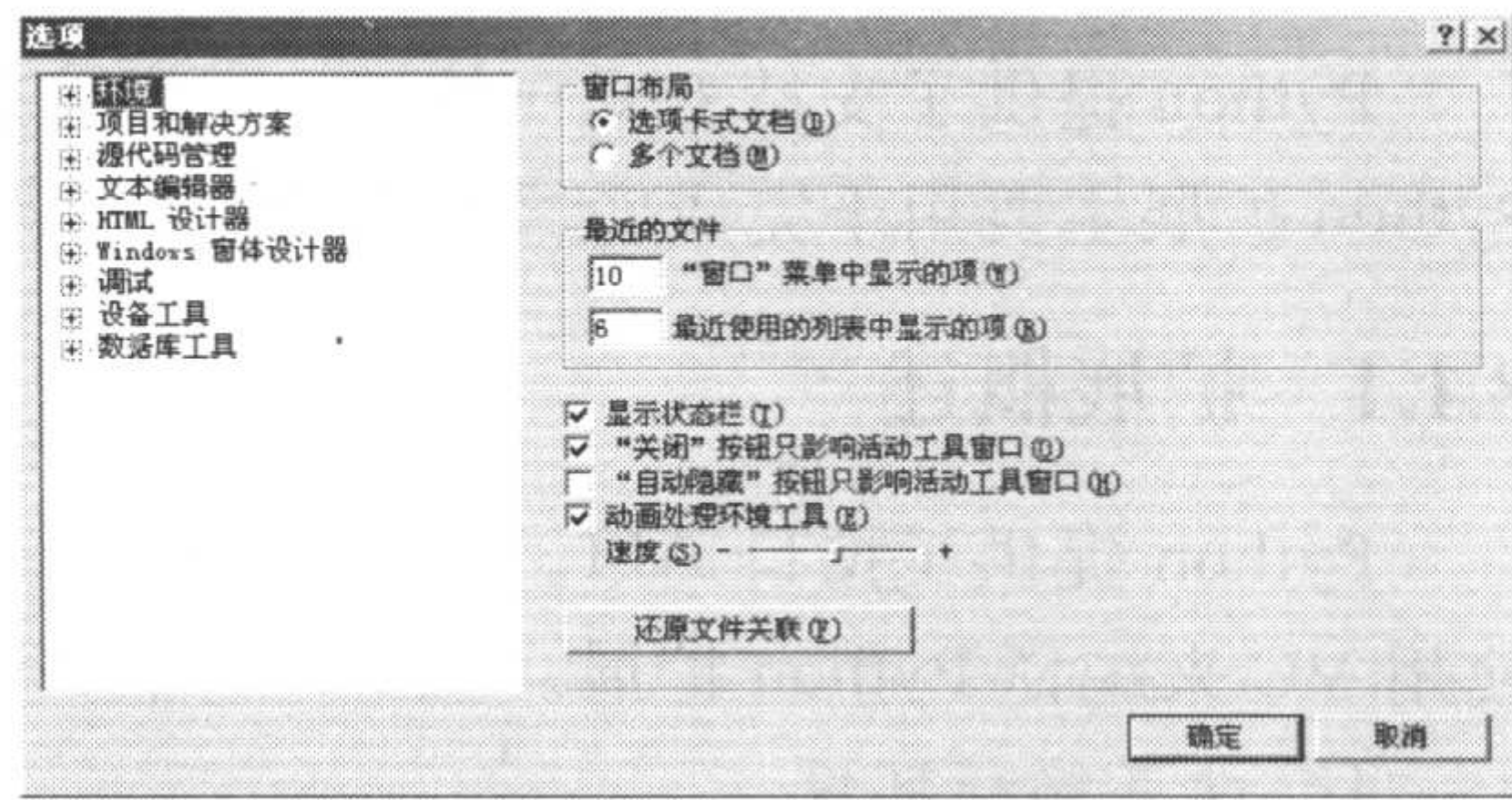


图 8-4 选项对话框

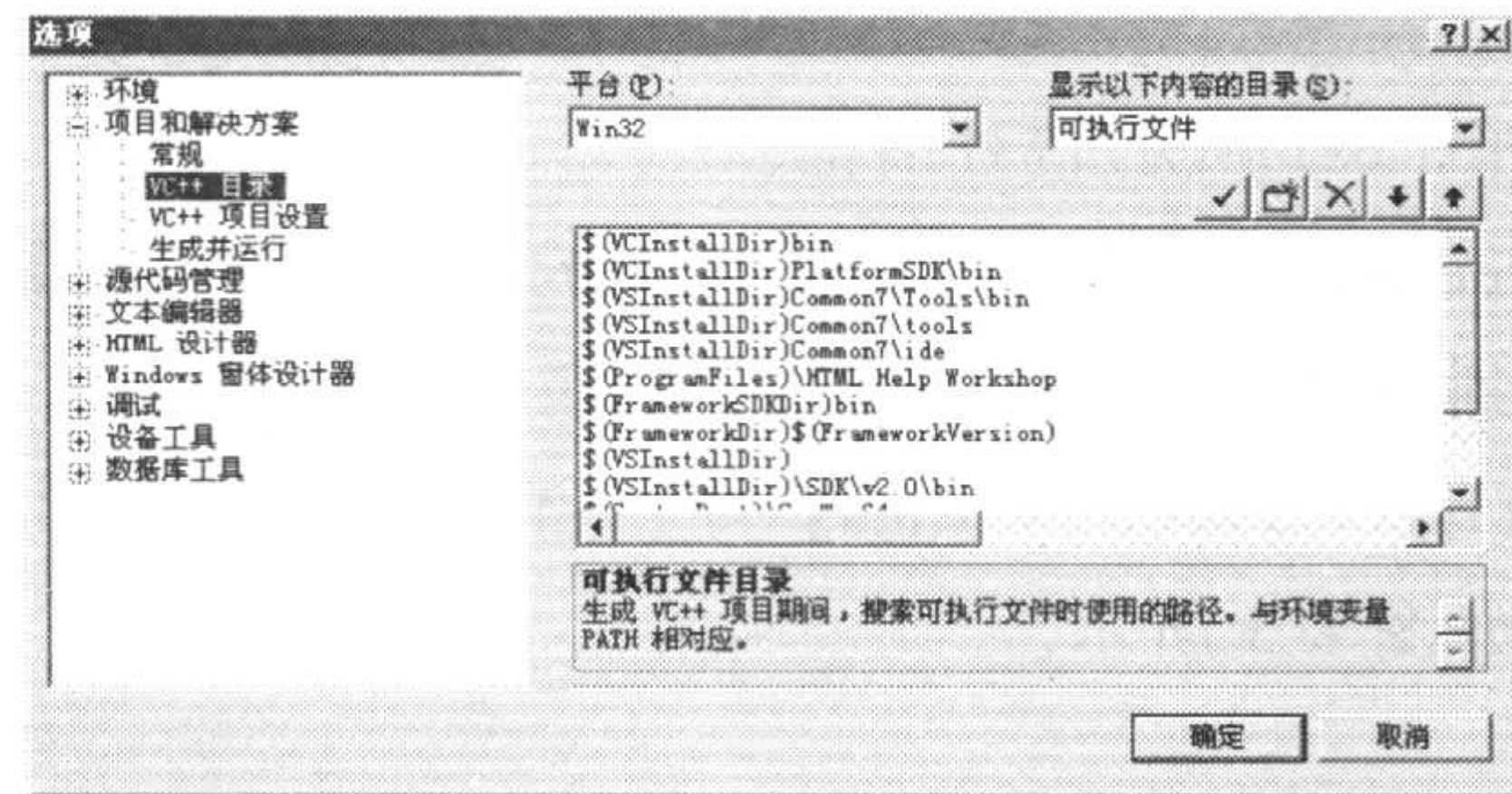


图 8-5 设置 VC++ 目录

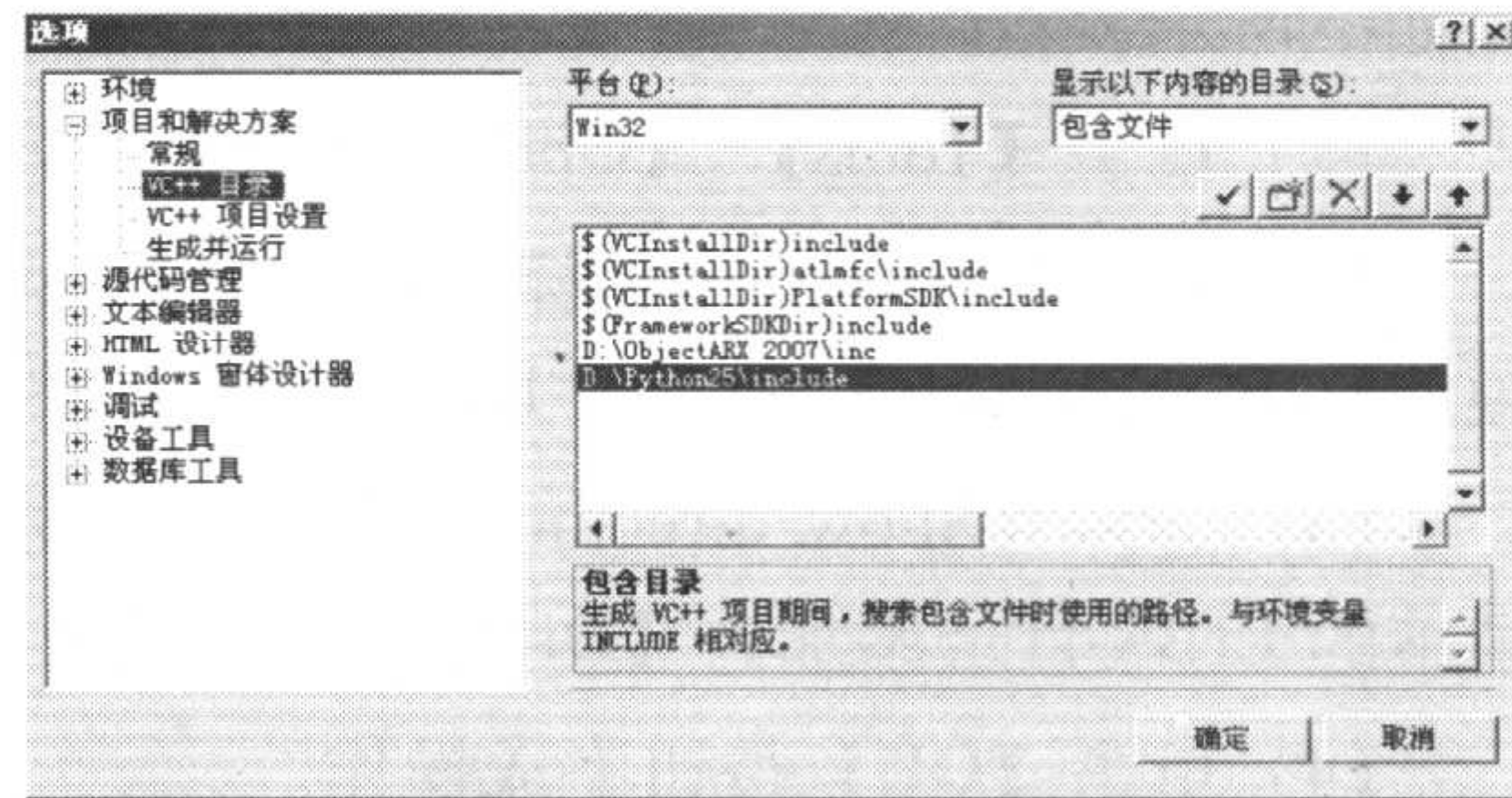


图 8-6 添加头文件

(4) 选择【显示以下内容的目录】下拉列表框中的【库文件】项，将 Python 安装目录下的 libs 目录添加到列表中，如图 8-7 所示。

(5) 单击【确定】按钮完成操作。

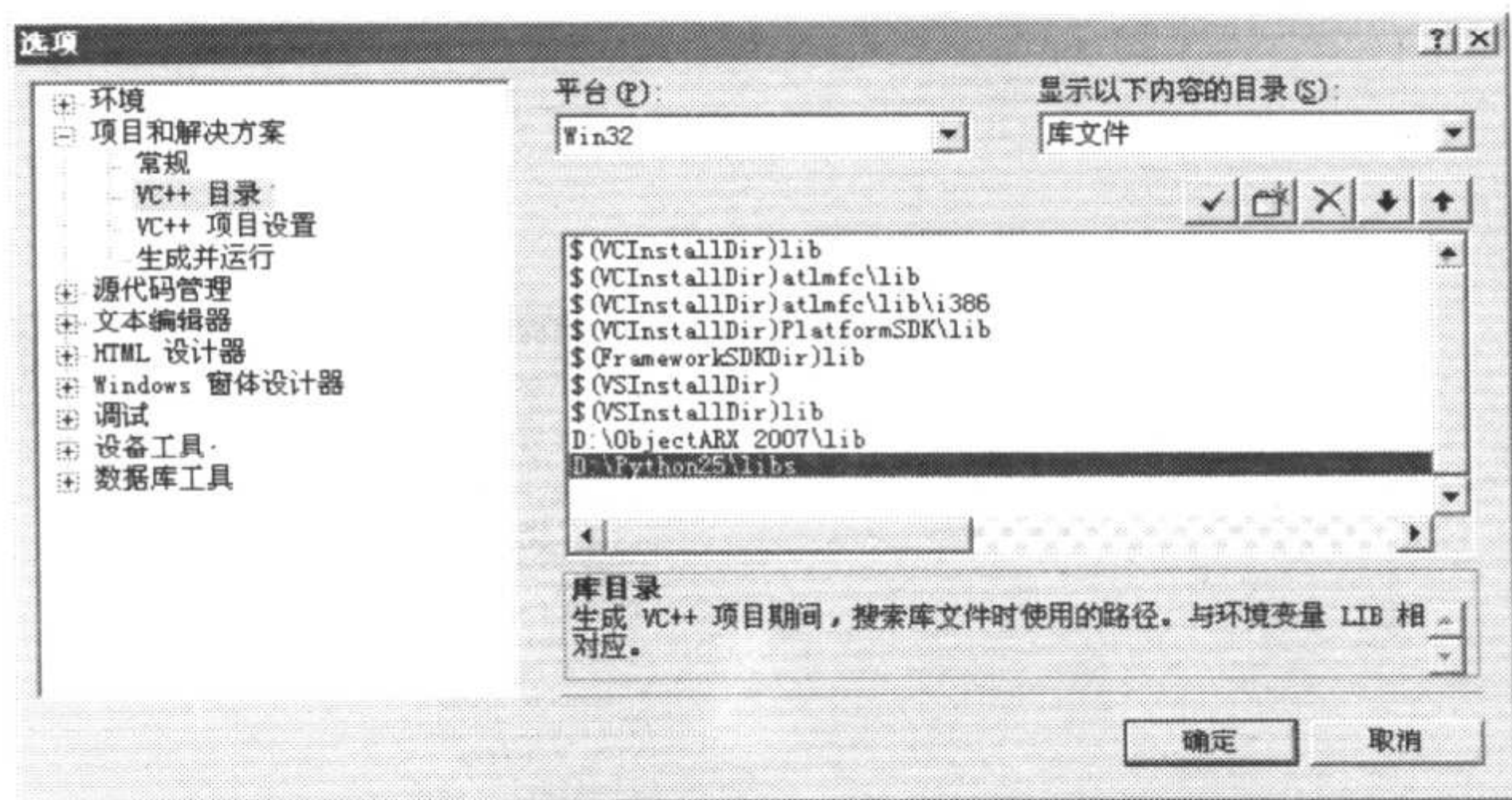


图 8-7 添加库文件

2. 创建工程

在 Visual C++ 6.0 中创建 Python 扩展过程如下所示。

(1) 单击 **【File】|【New】** 命令，弹出创建工程对话框。单击 **【Projects】** 标签，选择左侧列表中的 **【Win32 Dynamic-Link Library】** 项，在 **【Project name】** 文本框中输入工程名“myext”，如图 8-8 所示。

(2) 单击 **【OK】** 按钮，弹出如图 8-9 所示的工程设置对话框。选中 **【An empty DLL project.】** 单选框。

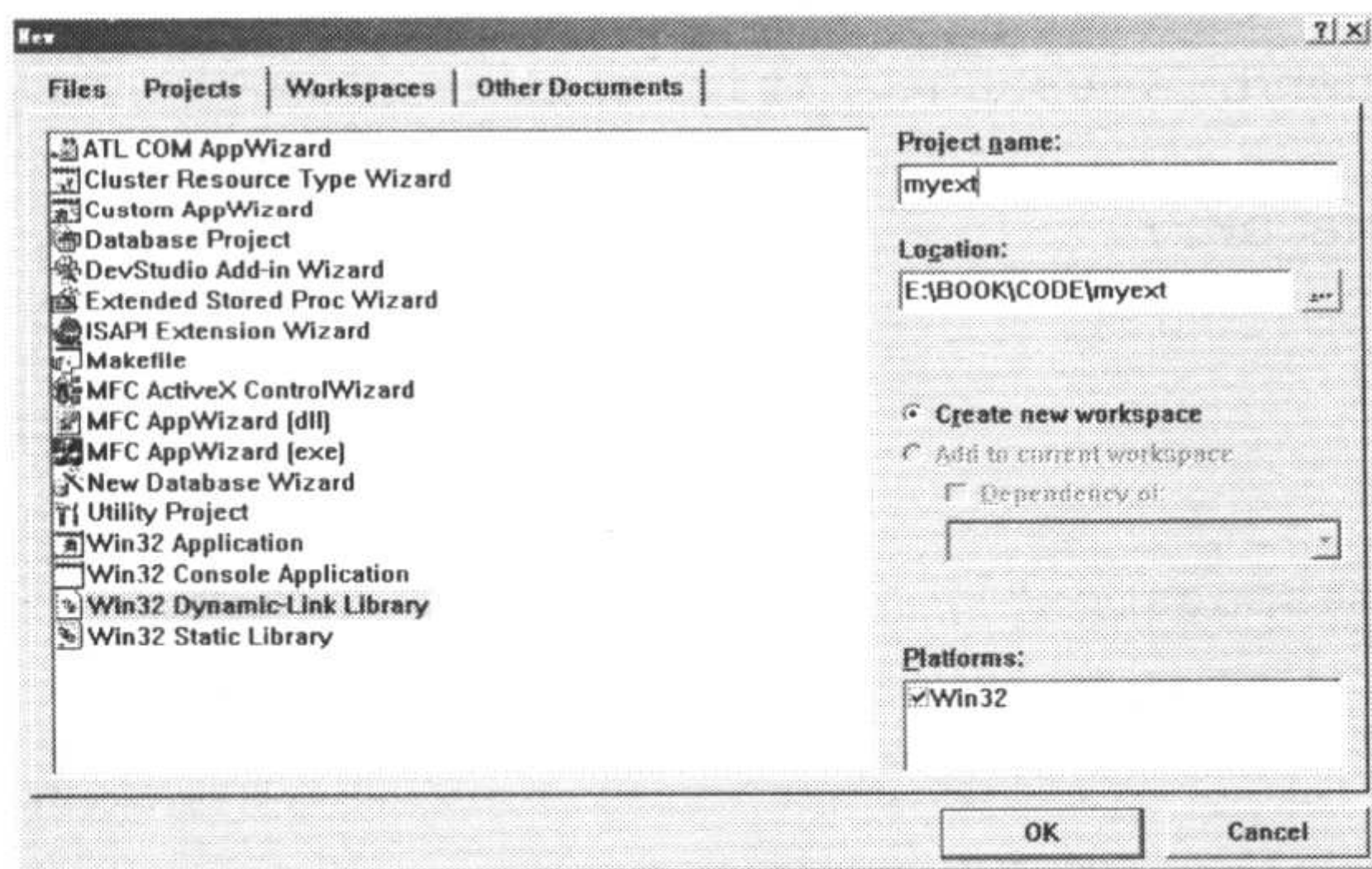


图 8-8 创建工程对话框

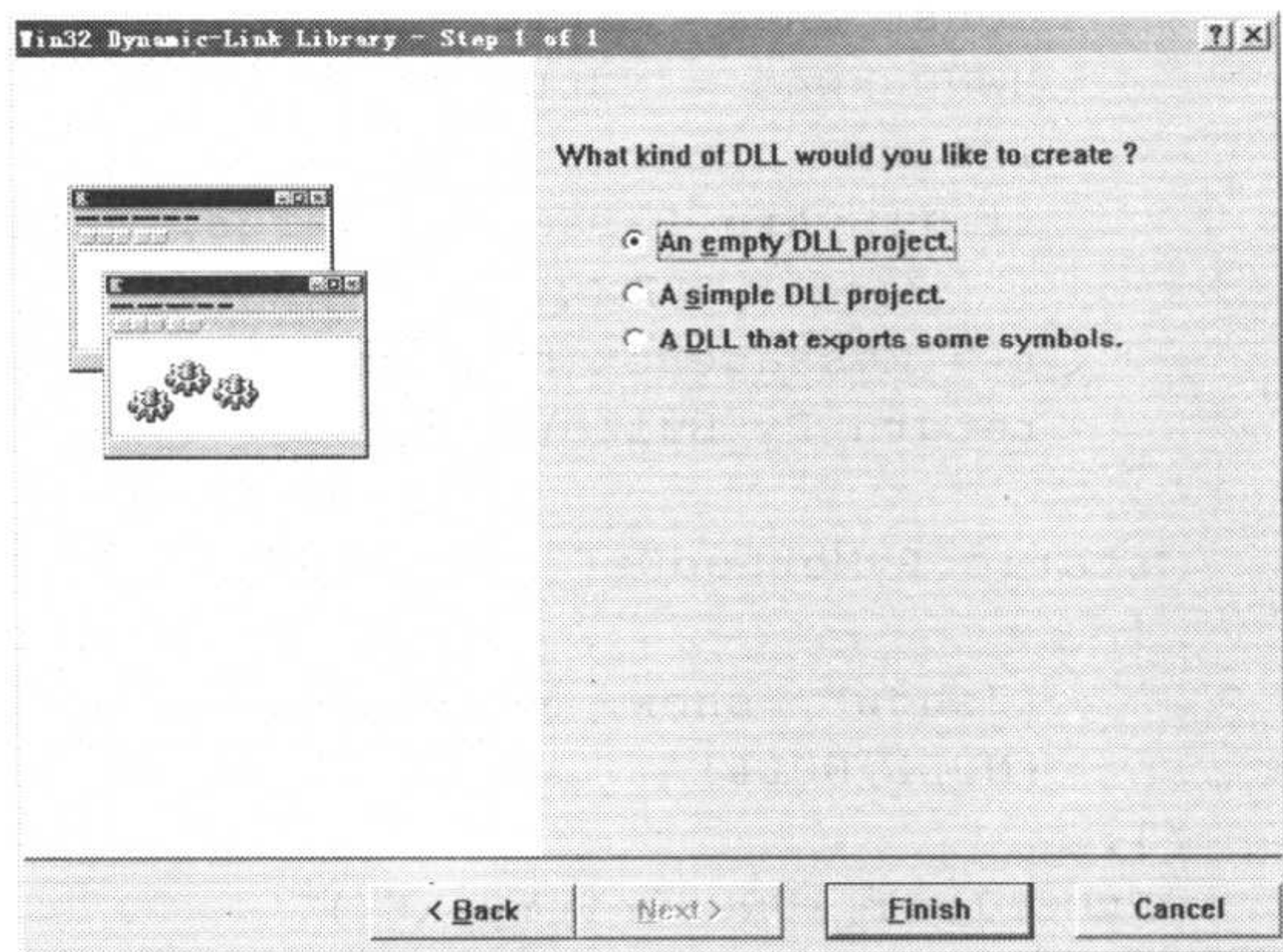


图 8-9 工程设置对话框

(3) 单击 **【Finish】** 按钮弹出如图 8-10 所示的确认对话框。单击 **【OK】** 按钮完成工程创建。

(4) 单击 **【File】|【New】** 命令弹出创建文件对话框，单击 **【Files】** 标签。选择左侧列表中的 **【C++ Source File】** 项，在 **【File】** 文本框中输入文件名“myext.c”，如图 8-11 所示。

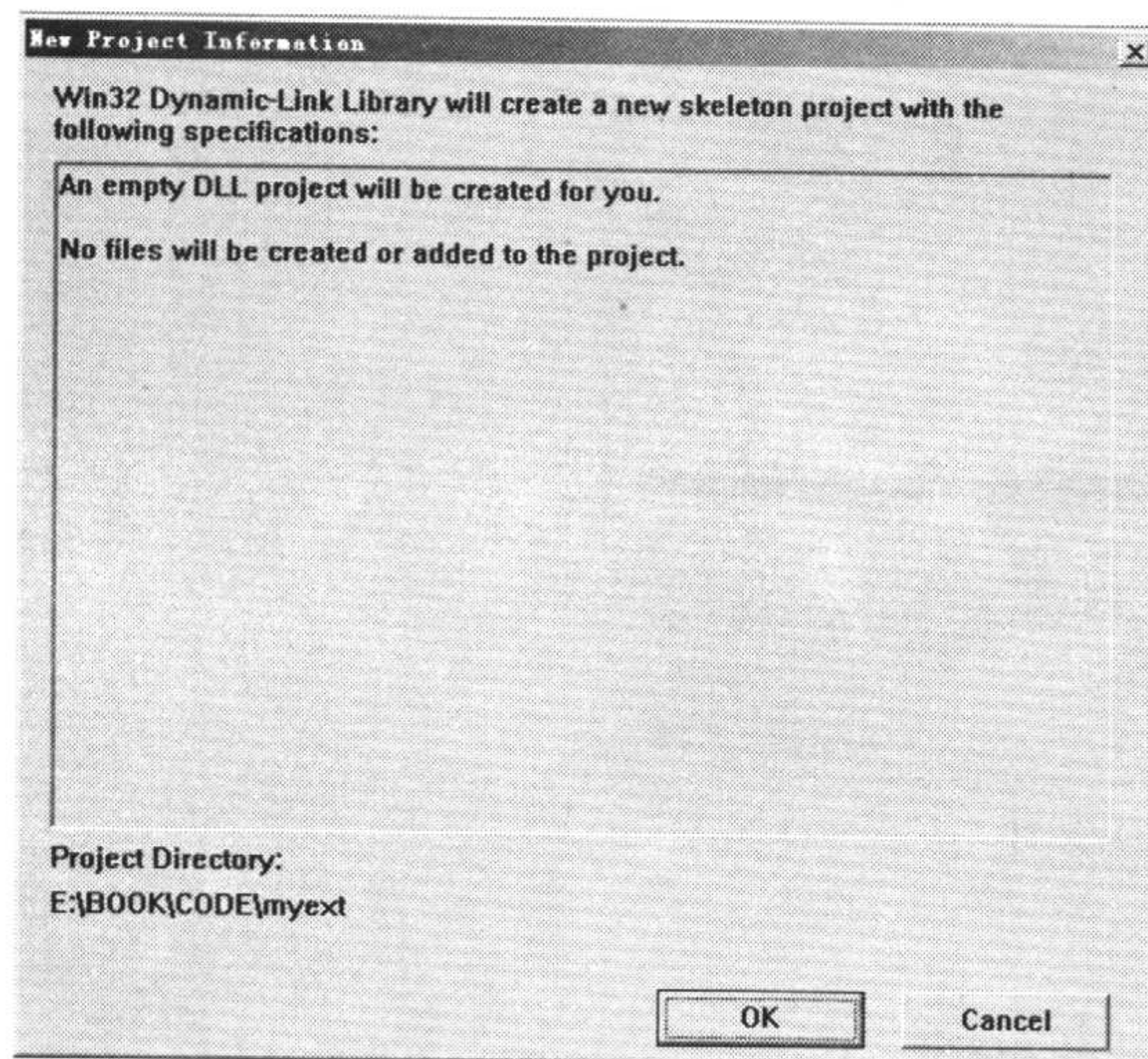


图 8-10 确认对话框

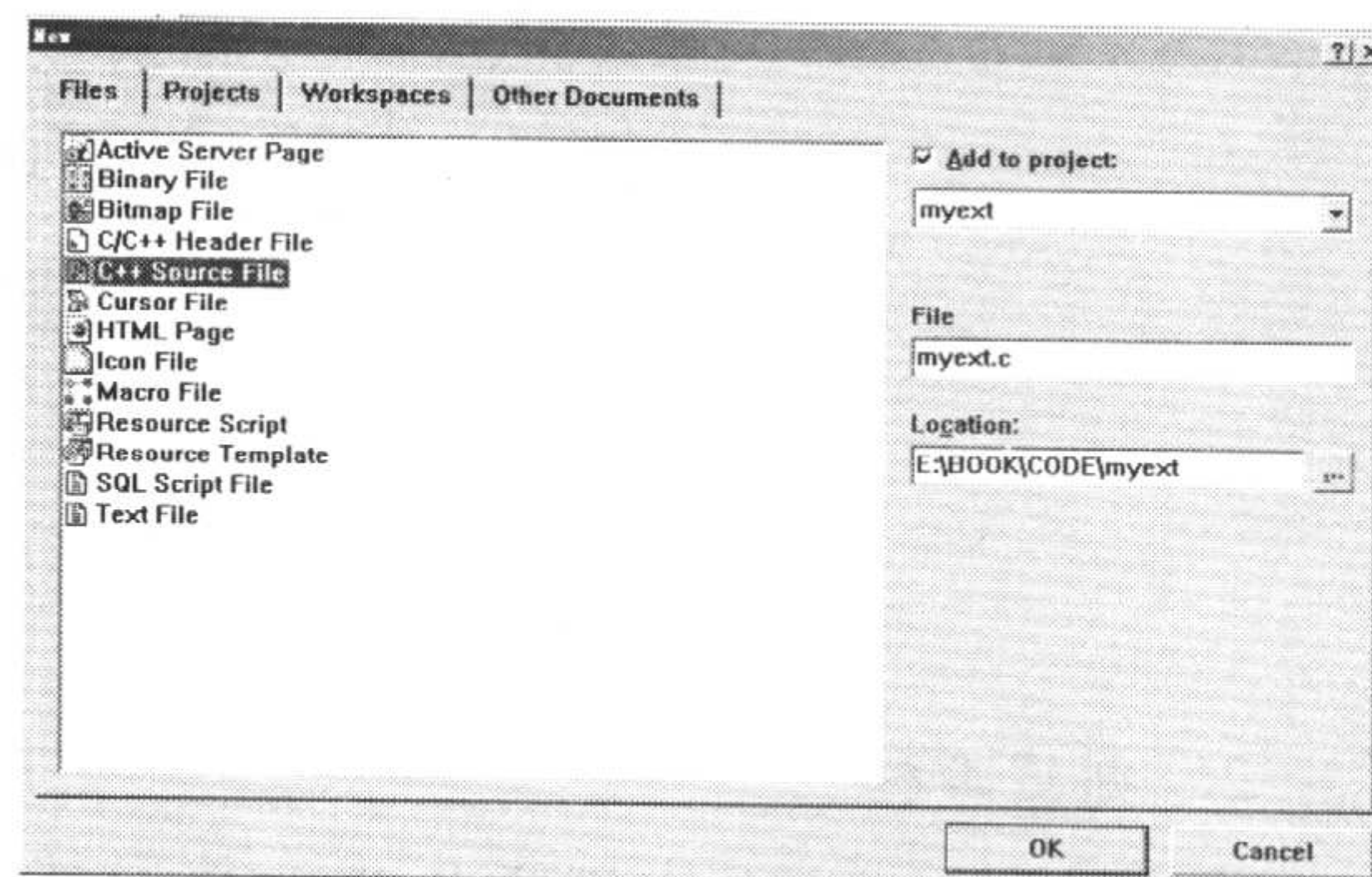


图 8-11 添加文件

(5) 单击 **【OK】** 按钮，在 “myext.c” 中添加如下内容。

```
#include <python.h>
#include <windows.h>
PyObject *show(PyObject *self, PyObject *args)
{
    char *message;
    const char *title = NULL;
    HWND hwnd = NULL;
    int r;
    if (!PyArg_ParseTuple(args, "iss", &hwnd, &message, &title))
        return NULL;
    r = MessageBox(hwnd, message, title, MB_OK);
    return Py_BuildValue("i", r);
}
static PyMethodDef myextMethods[] =
{
    {"show", show, METH_VARARGS, "show a messagebox"},
    {NULL, NULL}
};
PyMODINIT_FUNC initempty()
{
    PyObject *mod;
    mod = Py_InitModule("myext", myextMethods);
}
```

(6) 单击 **【Project】 | 【Settings】** 命令，弹出如图 8-12 所示的工程设置对话框。

(7) 选择 **【Settings For】** 下拉列表框中的 **【Win32 Release】** 项。单击 **【Link】** 标签，将 **【Output file name】** 文本框中的 “Release/myext.dll” 改为 “Release/myext.pyd”，如图 8-13 所示。

第8章 Python 扩展和嵌入

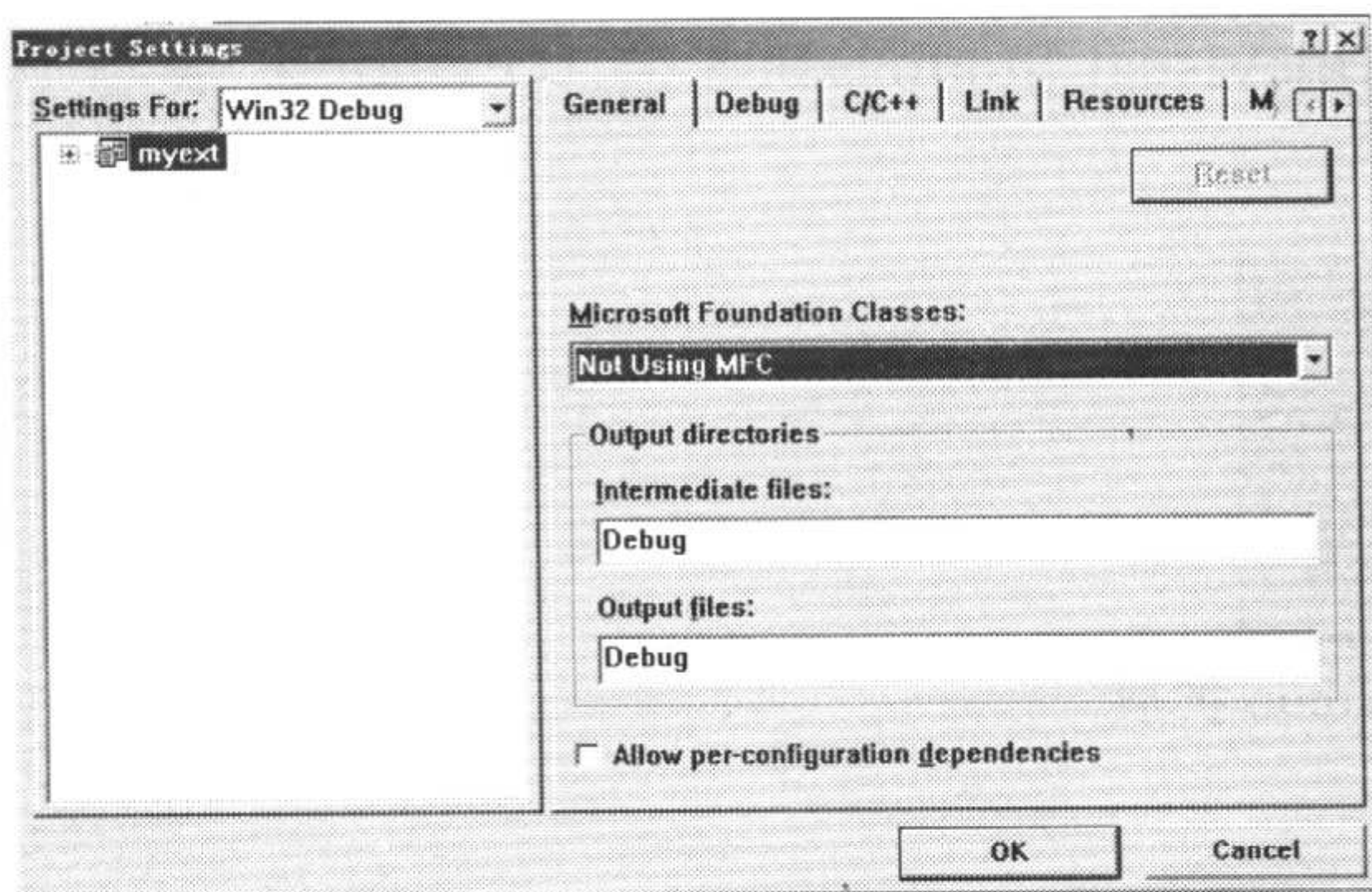


图 8-12 工程设置对话框

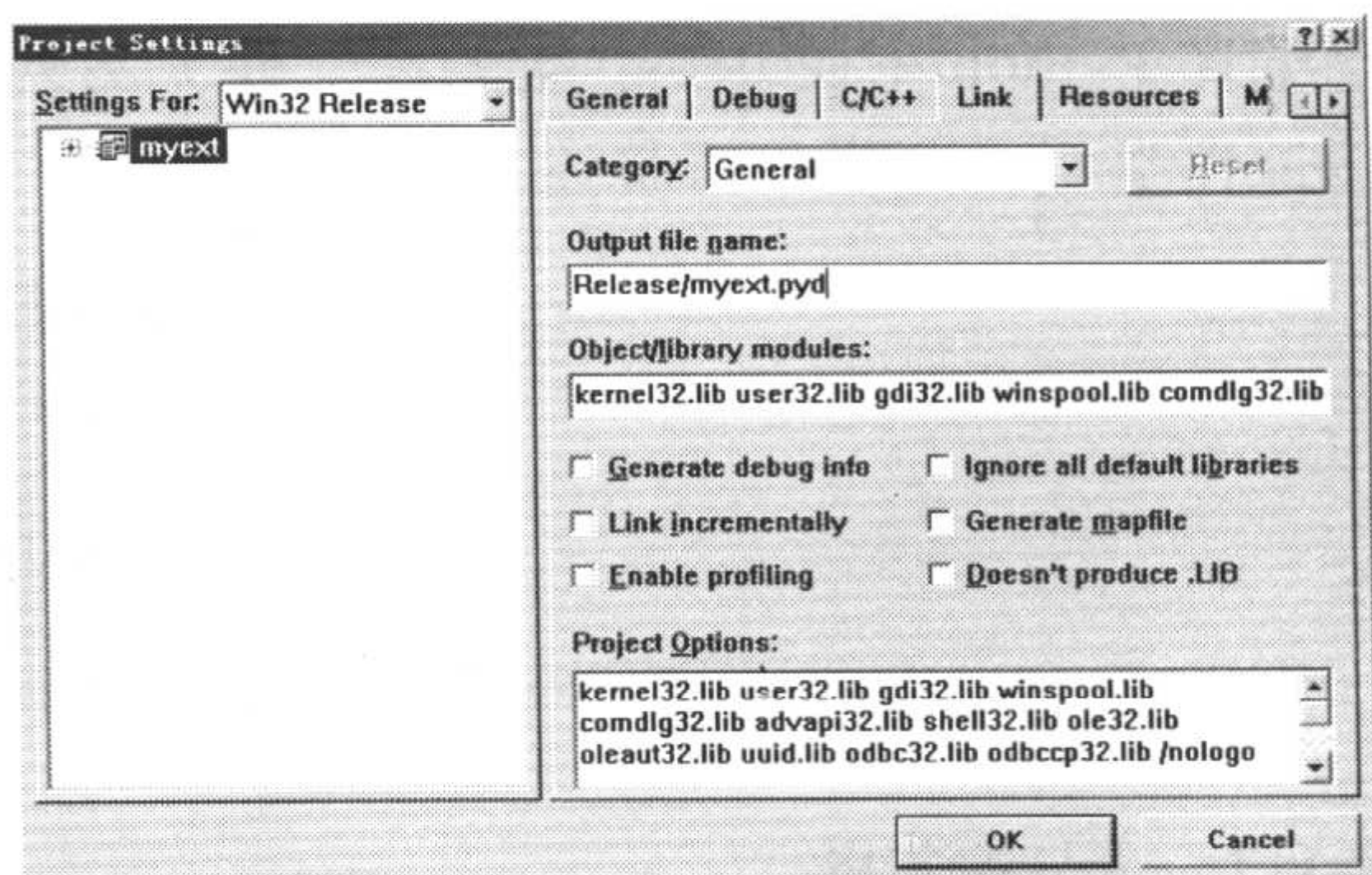


图 8-13 设置 Link 选项

(8) 单击【C/C++】标签，选择【Category】下拉列表框中的【Code Generation】项，选择【Use run-time library】下拉列表框中的【Multithreaded DLL】项，如图 8-14 所示。

(9) 单击【OK】按钮完成工程设置。

(10) 单击【Build】|【Batch Build】命令，弹出如图 8-15 所示的对话框，将【myext-Win32 Debug】单选框前的勾去掉。单击【Build】按钮生成 Python 扩展。

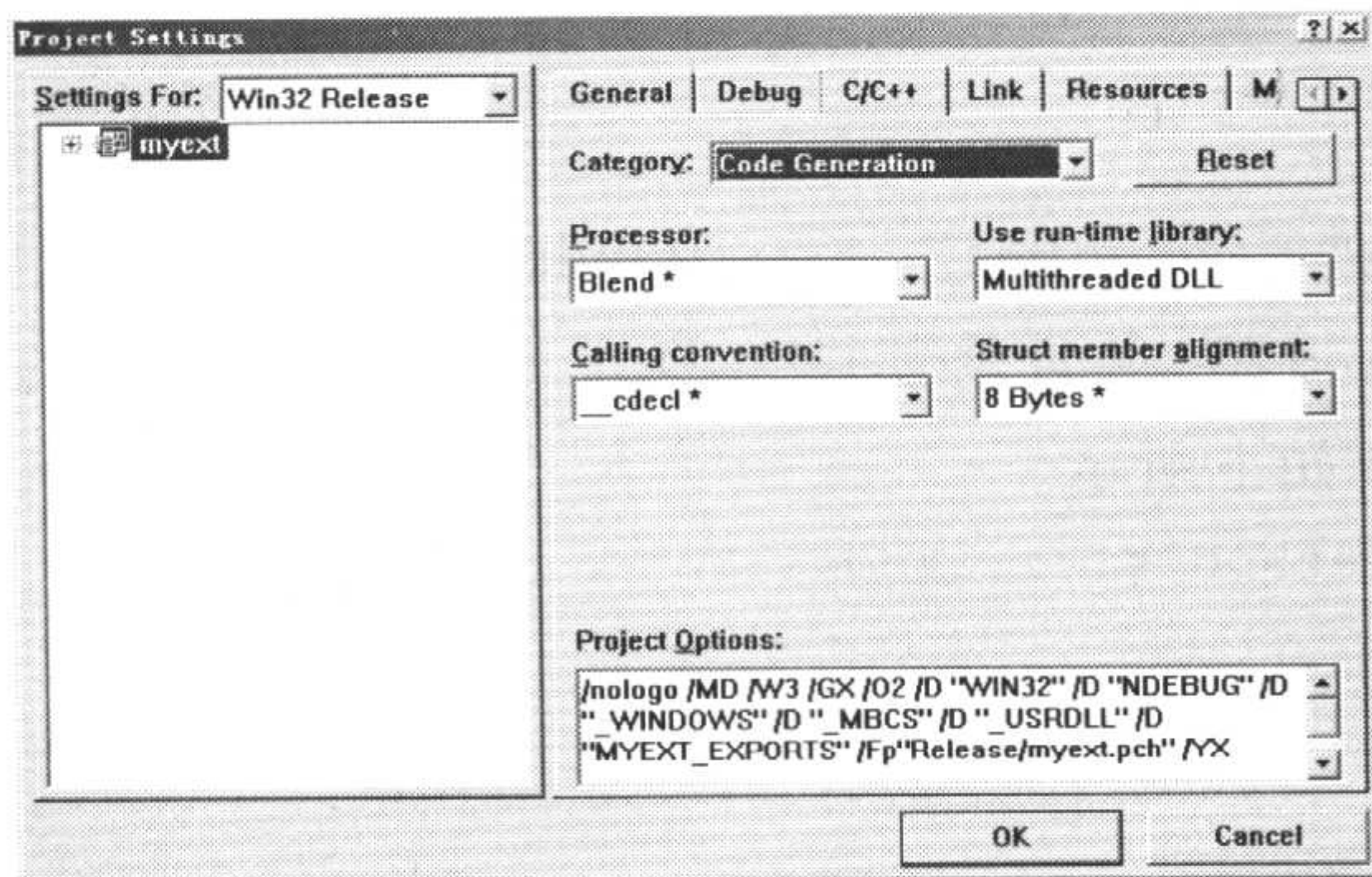


图 8-14 设置 C/C++选项

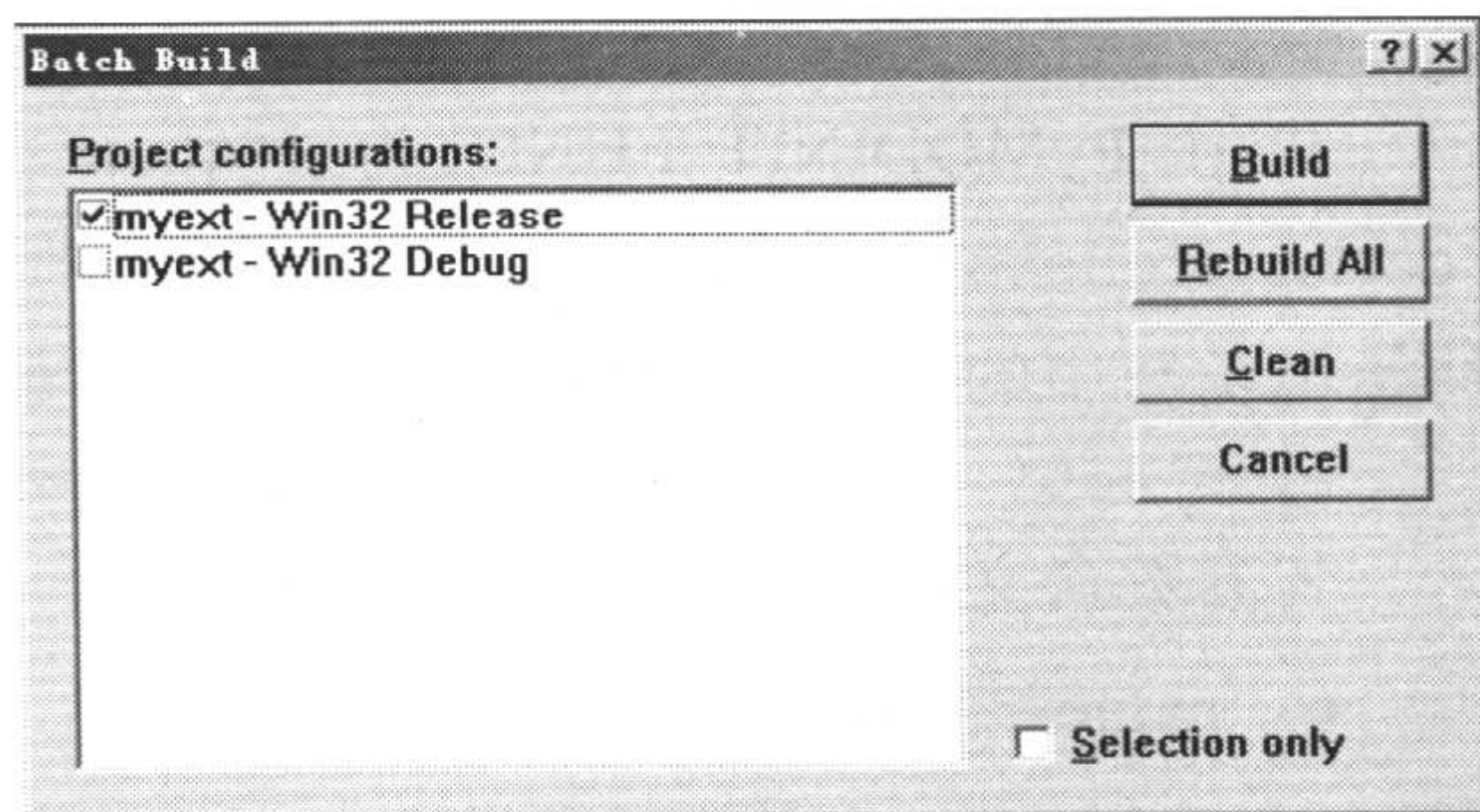


图 8-15 编译工程

编译完成后将在工程目录下的 Release 目录中生成“myext.pyd”文件，其为所编写的 Python 扩展。编写“usemyext.py”调用“myext.pyd”中的函数。代码如下所示。

```
# -*- coding:utf-8 -*-
# file: usemyext.py
#
import myext                                     # 导入 myext 模块
print myext.show(0, 'Extend Python', 'Python')  # 调用 show 函数
```

脚本运行后如图 8-16 所示。

由于 Python 官方的安装程序中不包含 debug 版的库文件，不能生成 debug 版的 Python 扩展，因此上述设置都是针对 release 版。如果需要生成 debug 版的 Python 扩展，则需要自

已编译 Python 生成 debug 版的库文件。另外，使用 Visual Studio 2005 编译 Python 扩展与上述过程类似，这里不再赘述。



图 8-16 使用 Python 扩展

8.1.2 程序详解

一般的 Python 扩展程序中应包含以下 3 部分内容。

1. 初始化函数

初始化函数是必须的，用于 Python 解释器对模块进行正确的初始化。初始化函数的函数名必须以 init 开头，并加上模块的名字。例如上节中初始化函数的函数名为“initmyext”，其中“myext”为模块名。函数“initmyext”代码如下所示。

```
PyMODINIT_FUNC initempty()
{
    PyObject *mod;
    mod = Py_InitModule("myext", myextMethods);
}
```

其中 PyMODINIT_FUNC 为 Python 头文件中定义的宏，在 Windows 下其相当于 _declspec(dllexport) void，即将 initempty 声明为 void 型，并且将其设为 DLL 文件的导出函数。初始化函数中的 Py_InitModule 函数，其函数原型如下所示。

```
PyObject* Py_InitModule( char *name, PyMethodDef *methods)
```

其参数含义如下。

- name: 模块名。
- methods: 方法列表。

2. 方法列表

方法列表中包含了 Python 扩展中的所有可以调用的函数方法。方法列表应该被声明为“static PyMethodDef”。上一节实例中的方法列表如下所示。

```
static PyMethodDef myextMethods[] =
{
    {"show", show, METH_VARARGS, "show a messagebox"},
    {NULL, NULL}
};
```

每一个函数方法对应于方法列表中的由大括号包围的一项。大括号中由 4 部分组成，模块中的方法名，与之对应的 Python 扩展中的函数名、函数调用方法，以及方法描述。其中函数调用方法应该为“METH_VARARGS”或者“METH_VARARGS | METH_KEYWORDS”。

也可以将函数调用方法设置为 0。方法列表应该以由两个 NULL 组成的一项来表示结束。

3. 函数实现

方法列表中包含了模块中方法对应的 C 语言函数实现。在 Python 扩展中所有的函数都应该被声明为 “PyObject*” 型，每个函数都应当含有两个 “PyObject*” 型的参数。在上一节的实例中，模块方法的实现函数如下所示。

```
PyObject *show(PyObject *self, PyObject *args)
{
    char *message;
    const char *title = NULL;
    HWND hwnd = NULL;
    int r;
    if (!PyArg_ParseTuple(args, "iss", &hwnd, &message, &title))
        return NULL;
    r = MessageBox(hwnd,message, title, MB_OK);
    return Py_BuildValue("i", r);
}
```

其中参数 self 只有在函数为 Python 的内置方法时才被使用，其余情况下 self 为一个空指针。参数 args 为在 Python 中向方法传递的参数。如果在方法列表中指定的函数调用方法为 “METH_VARARGS”，则在函数中使用 PyArg_ParseTuple 处理参数。如果在方法列表中指定的函数调用方法为 “METH_VARARGS | METH_KEYWORDS”，则应该使用 PyArg_ParseTupleAndKeywords 处理参数。其中 PyArg_ParseTuple 的函数原型如下所示。

```
int PyArg_ParseTuple( PyObject *args, const char *format, ...)
```

其参数含义如下。

- args：传递的参数。
- format：参数类型描述。

PyArg_ParseTuple 为可变参数函数，其后的参数即在函数中接受 Python 中传递参数的变量。在上述的 show 函数中，要使用 3 个参数，分别为 hwnd、message、title。在 PyArg_ParseTuple 中将其作为参数，使用 “&” 向 hwnd、message、title 传递值，即将 Python 向 show 方法传递的参数依次赋值给 hwnd、message、title。

PyArg_ParseTuple 函数中的 format 参数指定了其后续参数的类型，在 show 函数中 format 参数为 “iss” 表示 hwnd 为整型，message 和 title 为字符串。常见的指定参数类型的字符如表 8-1 所示。

表 8-1 常见的指定参数类型的字符

格式化字符	C 数据类型	Python 类型
s	char*	字符串
s#	char*, int	字符串及长度
z	char*	与 s 相同，但可以为 NULL

续表

格式化字符	C 数据类型	Python 类型
z#	char*, int	与 s#相同，但可以为 NULL
i	int	长整型
l	long int	长整型
c	char	单个字符的字符串
f	float	双精度型
d	double	双精度型

8.1.3 在 Python 扩展中使用 MFC

在 Windows 下使用 MFC 可以方便地进行 GUI 编程。MFC 对基本的 SDK API 函数进行了封装，使用更为简便。在 PythonWin 中提供了部分 MFC 中的函数。在 Python 扩展中使用 MFC 与上一节中的例子有不同的地方。此处给出一个在 Python 扩展中使用 MFC 创建一个对话框的例子。整个过程如下所示。

(1) 单击【File】|【New】命令，弹出创建工程对话框。单击【Projects】标签，选择左侧列表中的【MFC AppWizard (dll)】项，在【Project name】文本框中输入工程名“UseMFC”，如图 8-17 所示。

(2) 单击【OK】按钮，弹出如图 8-18 所示的工程设置对话框。选中【Regular DLL using shared MFC DLL】单选框，使用动态链接方式。该方式需要 MFC DLL 的支持，如果选中【Regular DLL with MFC statically linked】单选框，则使用静态链接的方式，这样会增大生成的 Python 扩展的体积。

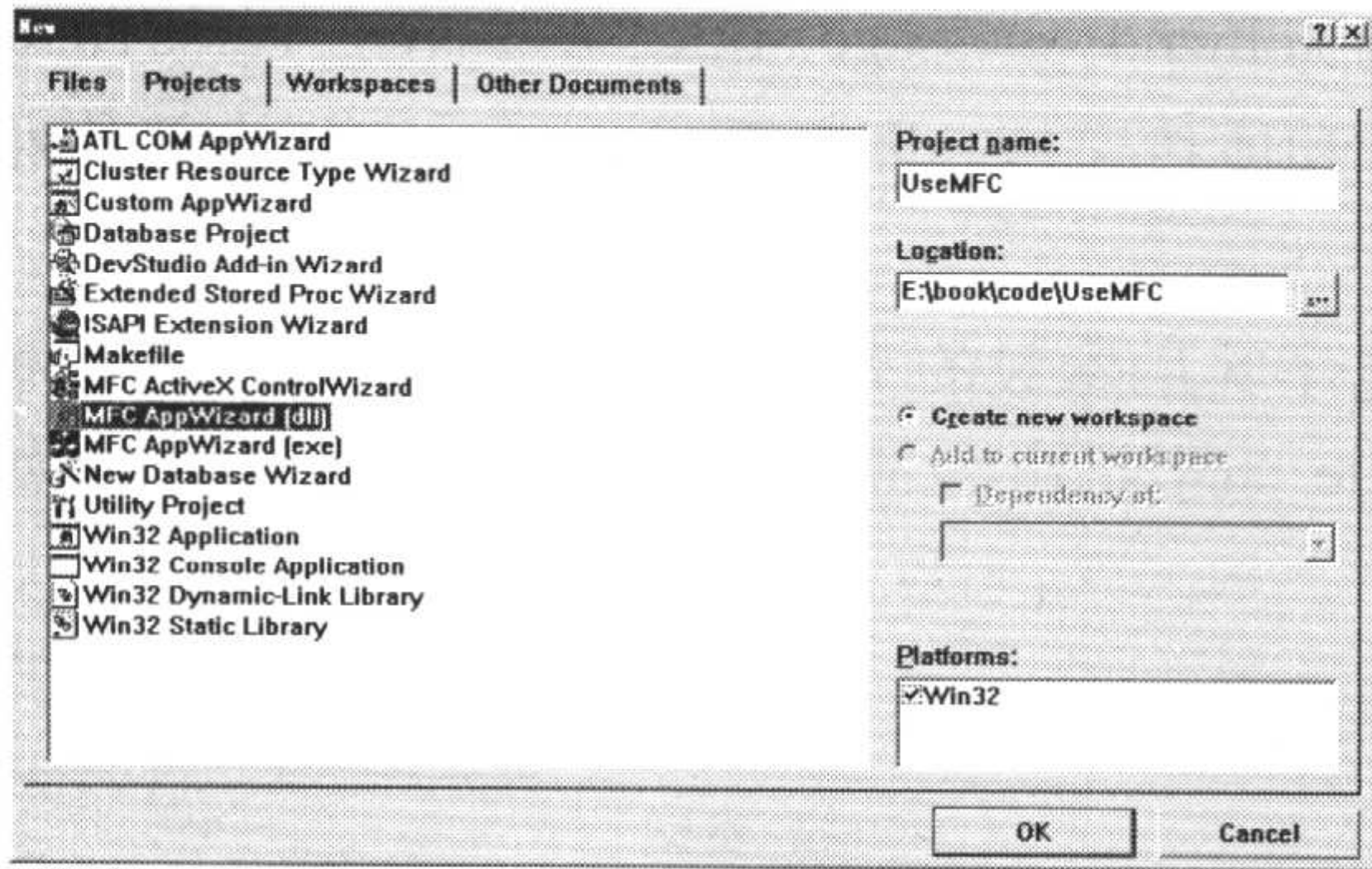


图 8-17 创建工程对话框

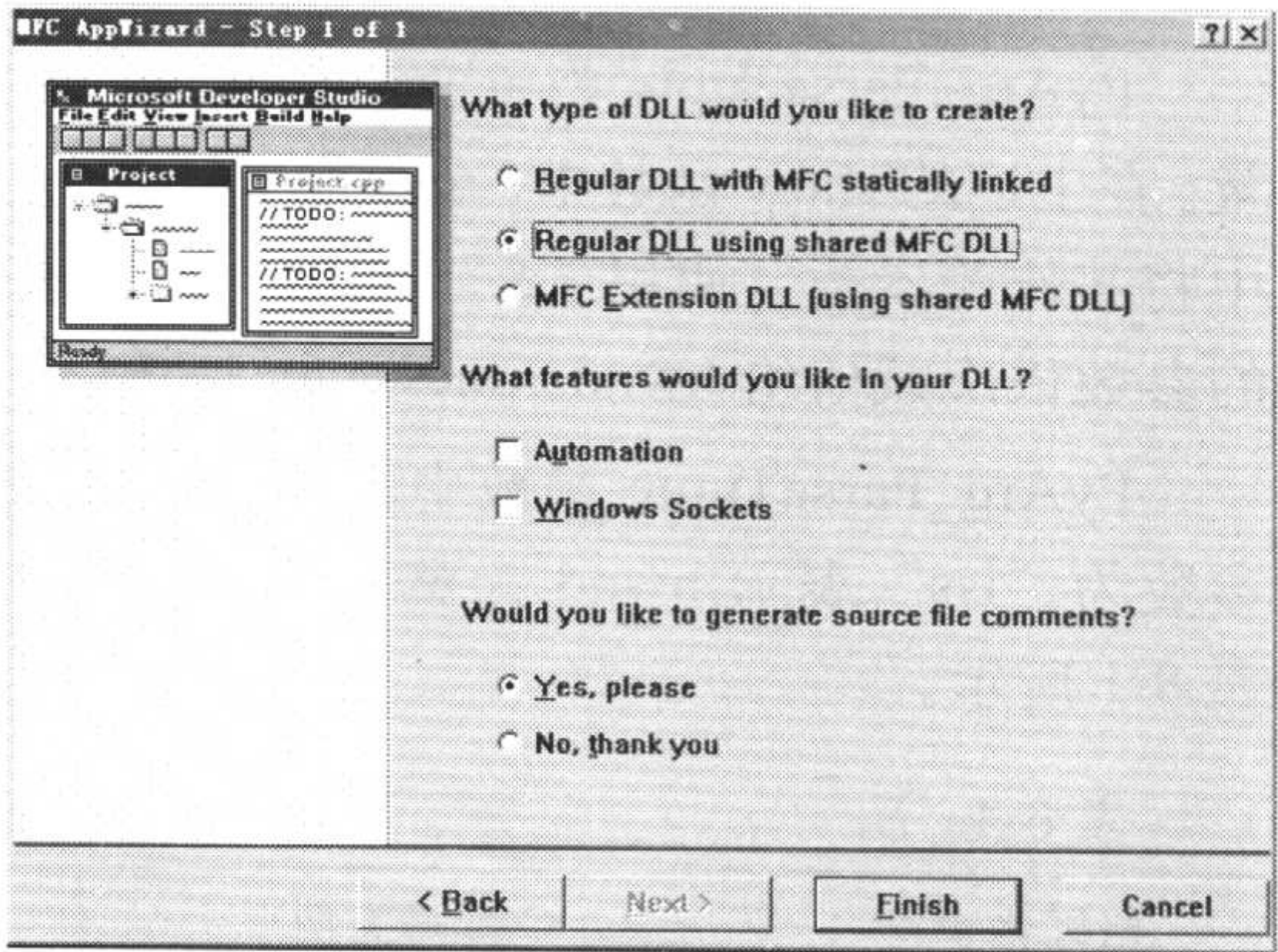


图 8-18 设置工程类型对话框

(3) 单击【Finish】按钮，弹出如图 8-19 所示的确认对话框。单击【OK】按钮完成工程创建。

(4) 单击 **【Insert】|【Resource】** 命令，弹出添加资源对话框，选择左侧列表中的 **【Dialog】** 项，如图 8-20 所示。

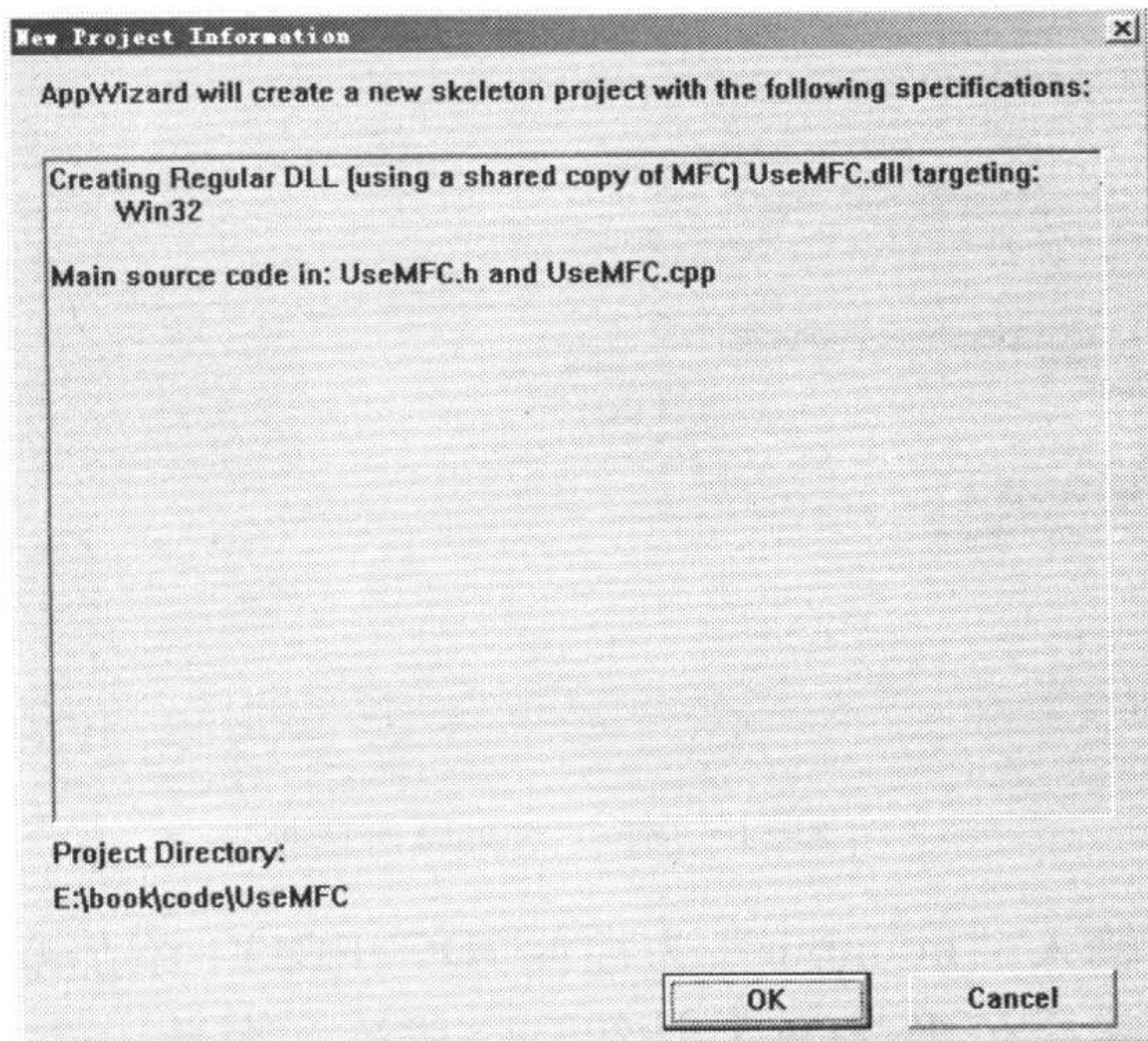


图 8-19 工程信息确认对话框

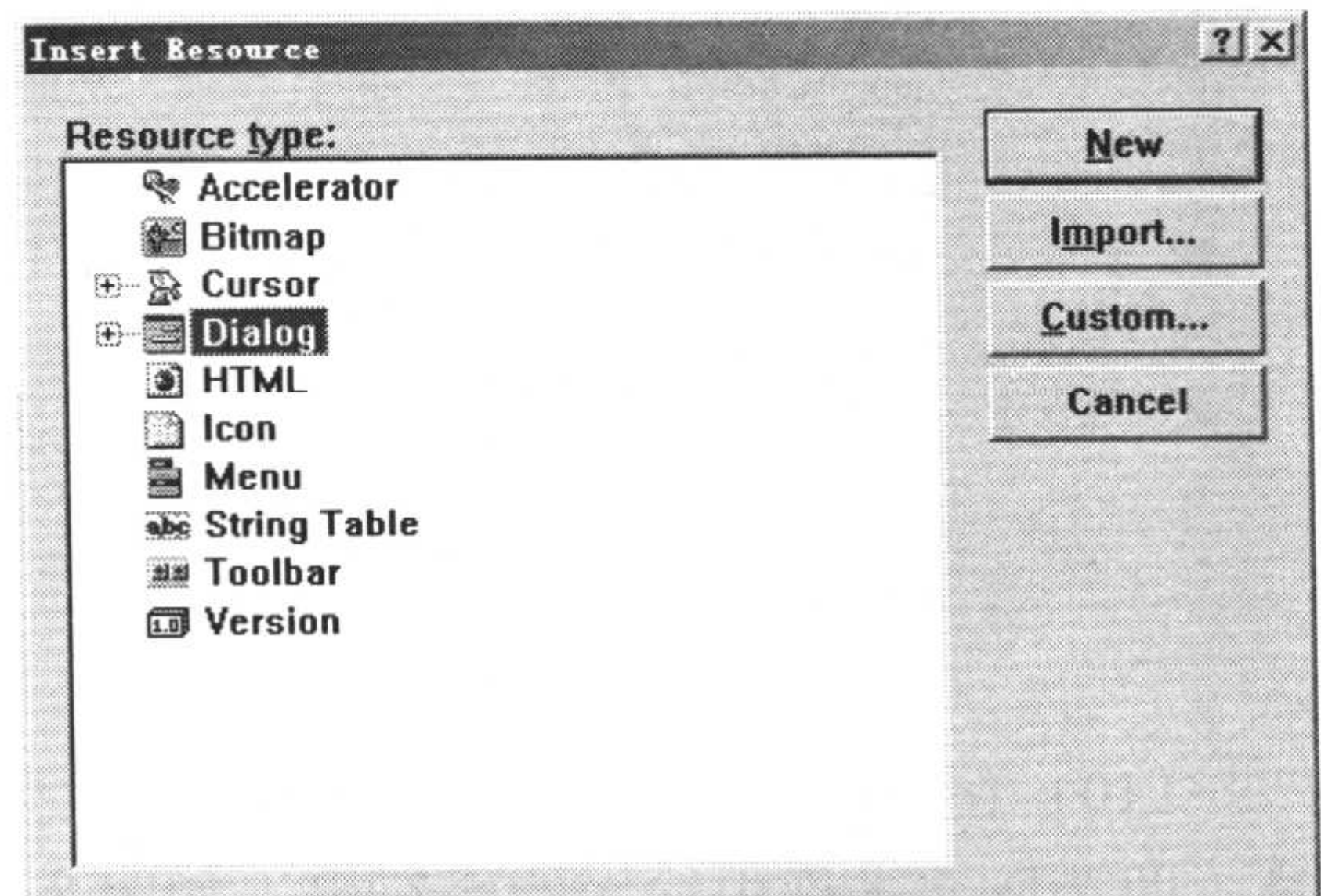


图 8-20 添加资源对话框

(5) 单击 **【New】** 按钮将在工程中新建一个对话框。向对话框中添加 Edit 控件和 Static Text 控件，将其修改为如图 8-21 所示的形式。

(6) 在创建的对话框上右击，选择 **【ClassWizard】** 命令，弹出如图 8-22 所示的添加类对话框。

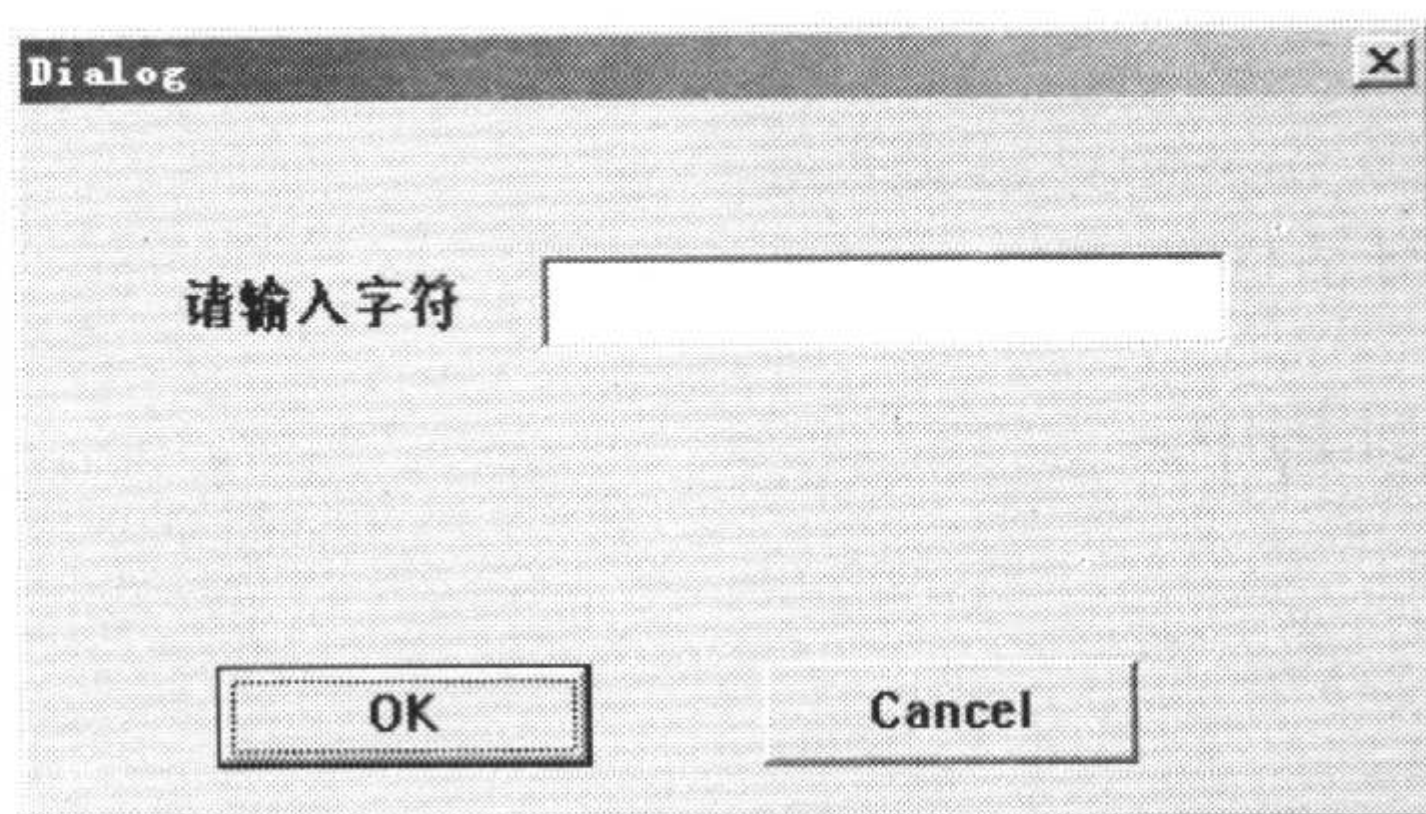


图 8-21 创建对话框

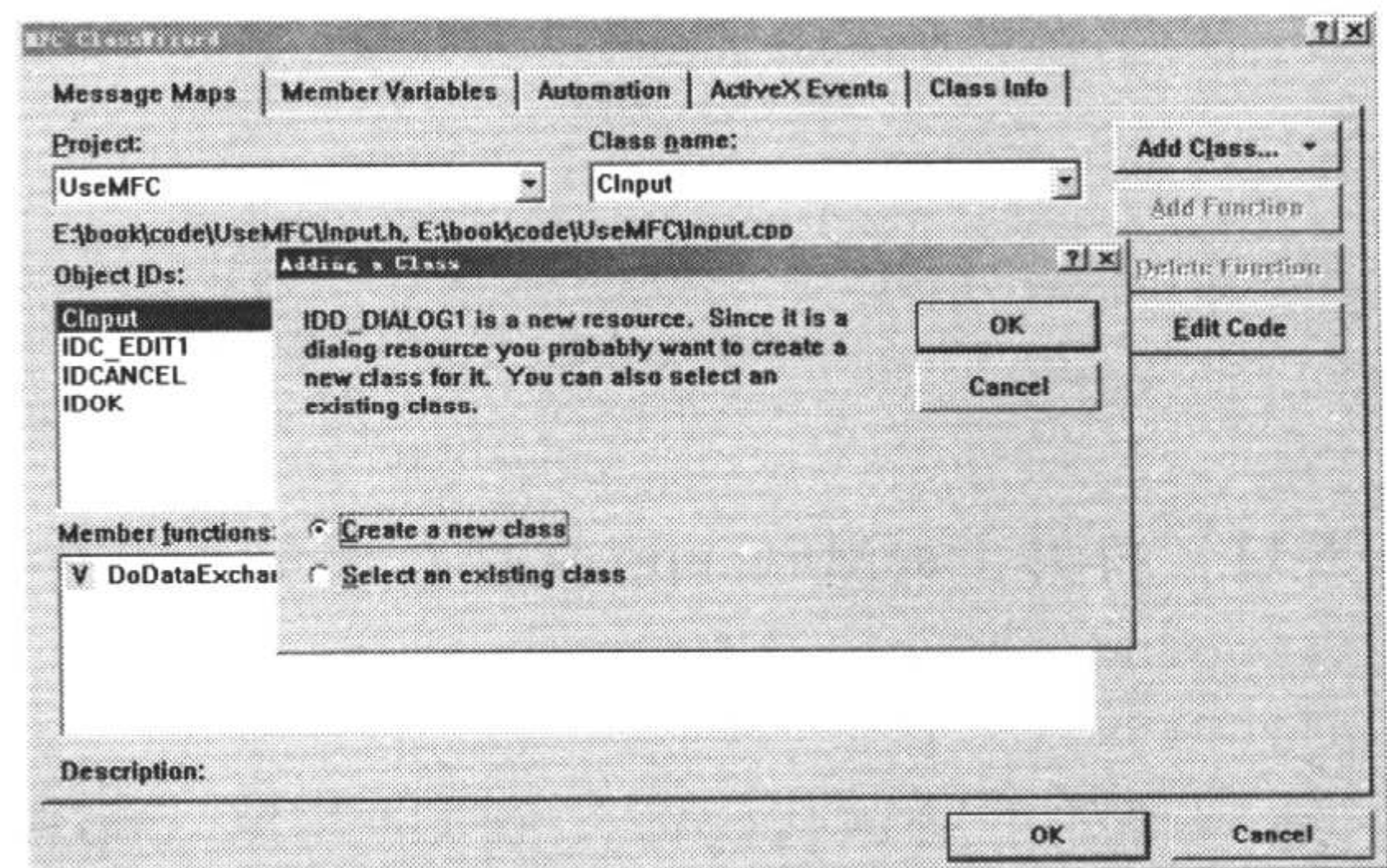


图 8-22 为对话框添加类

(7) 单击 **【OK】** 按钮为对话框添加一个新类，在弹出的添加类对话框中的 **【Name】** 文本框中将类命名为“CInput”，其余按照默认选项，如图 8-23 所示。

(8) 右击创建的对话框，选择 **【ClassWizard】** 命令，弹出如图 8-24 所示的对话框。

(9) 单击 **【Member Variables】** 标签，选中 **【IDC_EDIT1】** 项，单击 **【Add Variable】** 按

钮，弹出如图 8-25 所示的对话框。

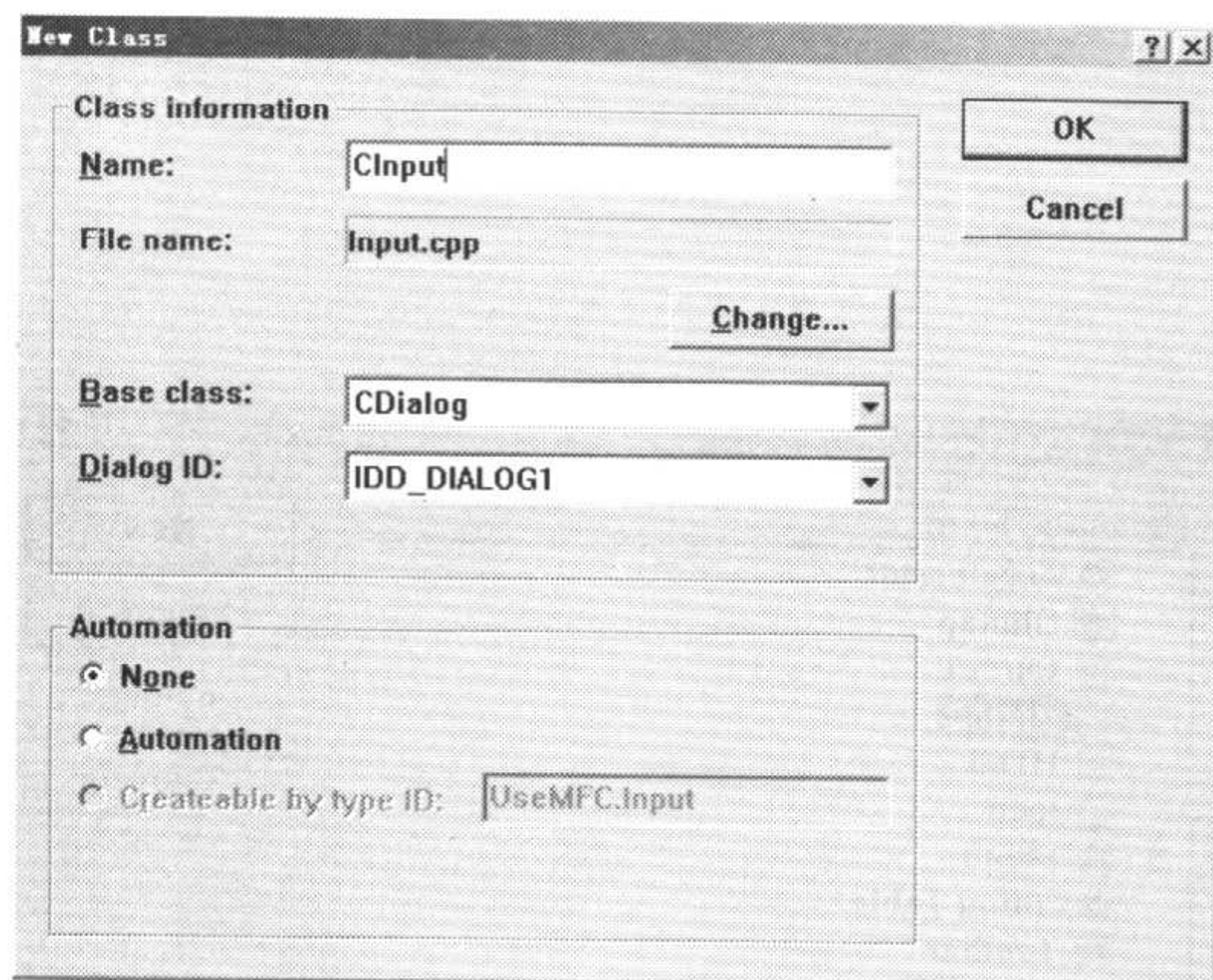


图 8-23 输入类名

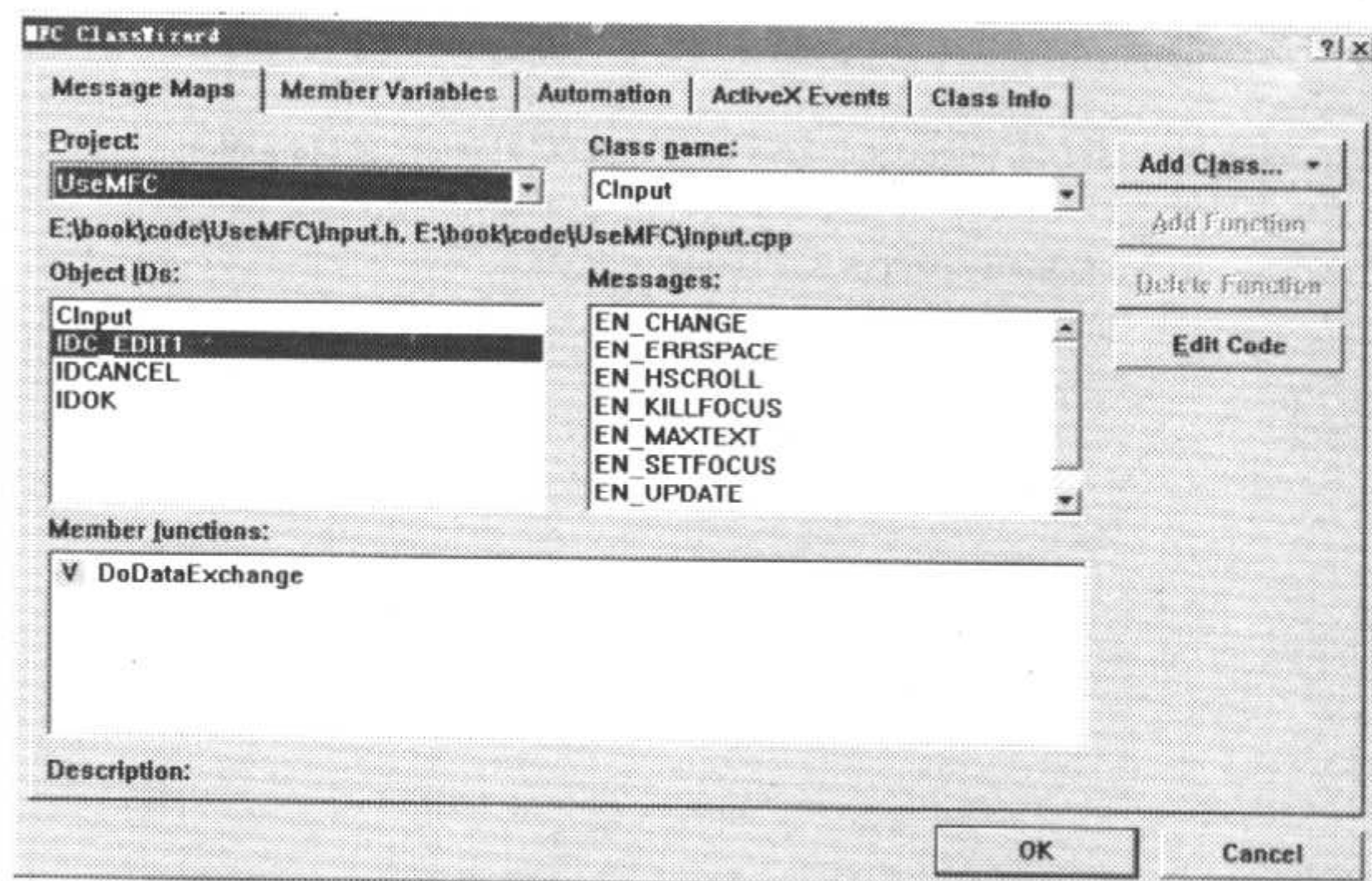


图 8-24 MFC 类向导对话框

(10) 在【Member variable name】文本框中输入“m_input”为控件 IDC_EDIT1 添加变量，即获取文本框中输入的字符串，如图 8-26 所示。单击【OK】按钮，完成添加变量。

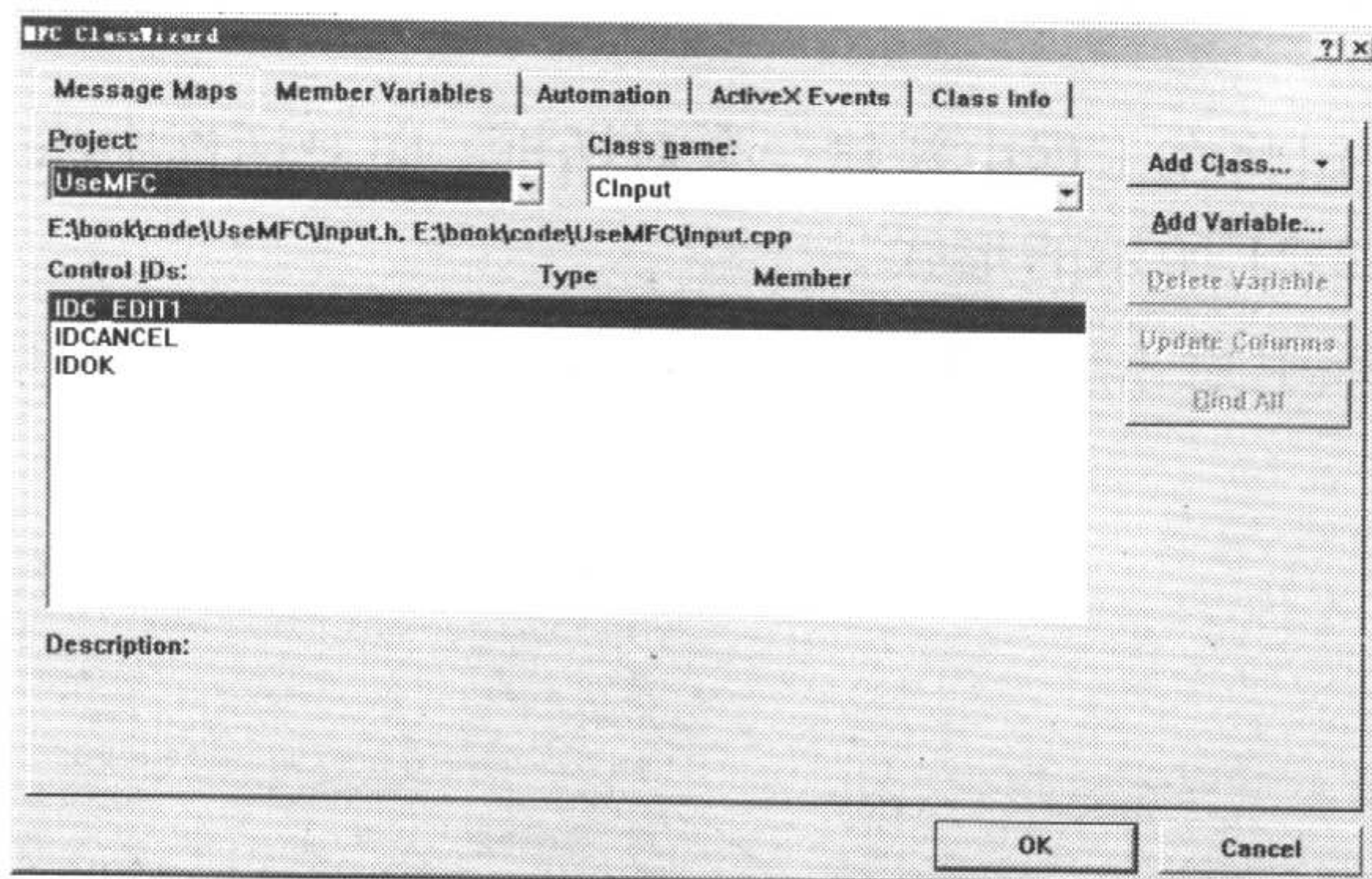


图 8-25 添加变量对话框

(11) 打开 UseMFC.cpp 文件，将如下文件添加到其中。

```
#include "Input.h"
#include <Python.h>
```

然后将如下所示代码添加到 UseMFC.cpp 文件中。

```
PyObject *show(PyObject *self, PyObject *args)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    CInput dia;
    dia.DoModal();
    return Py_BuildValue("s", dia.m_input);
}

static PyMethodDef UseMFCMethods[] =
```



```
{
    {"show", show, METH_VARARGS, "show a messagebox"},
    {NULL, NULL}
};
extern "C" void initUseMFC()
{
    PyObject *mod;
    mod = Py_InitModule("UseMFC", UseMFCMethods);
}
```

(12) 打开 UseMFC.def 文件，将初始化函数添加到 UseMFC.def 文件中。def 文件是用来告诉链接器 DLL 文件的导出函数的，相当于使用 PyMODINIT_FUNC 声明初始化函数。UseMFC.def 文件内容如下所示。

; UseMFC.def : Declares the module parameters for the DLL.

```
LIBRARY      "UseMFC"
DESCRIPTION  'UseMFC Windows Dynamic Link Library'
```

```
EXPORTS
    ; Explicit exports can go here
    initUseMFC
```

(13) 按照 8.1.1 节中创建工程的第 (6) ~ (10) 步操作，完成 Python 扩展的编译。

(14) 编写如下所示的 UseMFC.py，调用编译好的 UseMFC 模块。

```
# -*- coding:utf-8 -*-
# file: UseMFC.py
#
import UseMFC                                # 导入 UseMFC 模块
input = UseMFC.show()                        # 调用 show 函数
print '刚才输入的是: '
print input
```

(15) 运行脚本后，在文本框中输入“Hi,Python and MFC!”，如图 8-27 所示。单击【OK】按钮后，如图 8-28 所示。

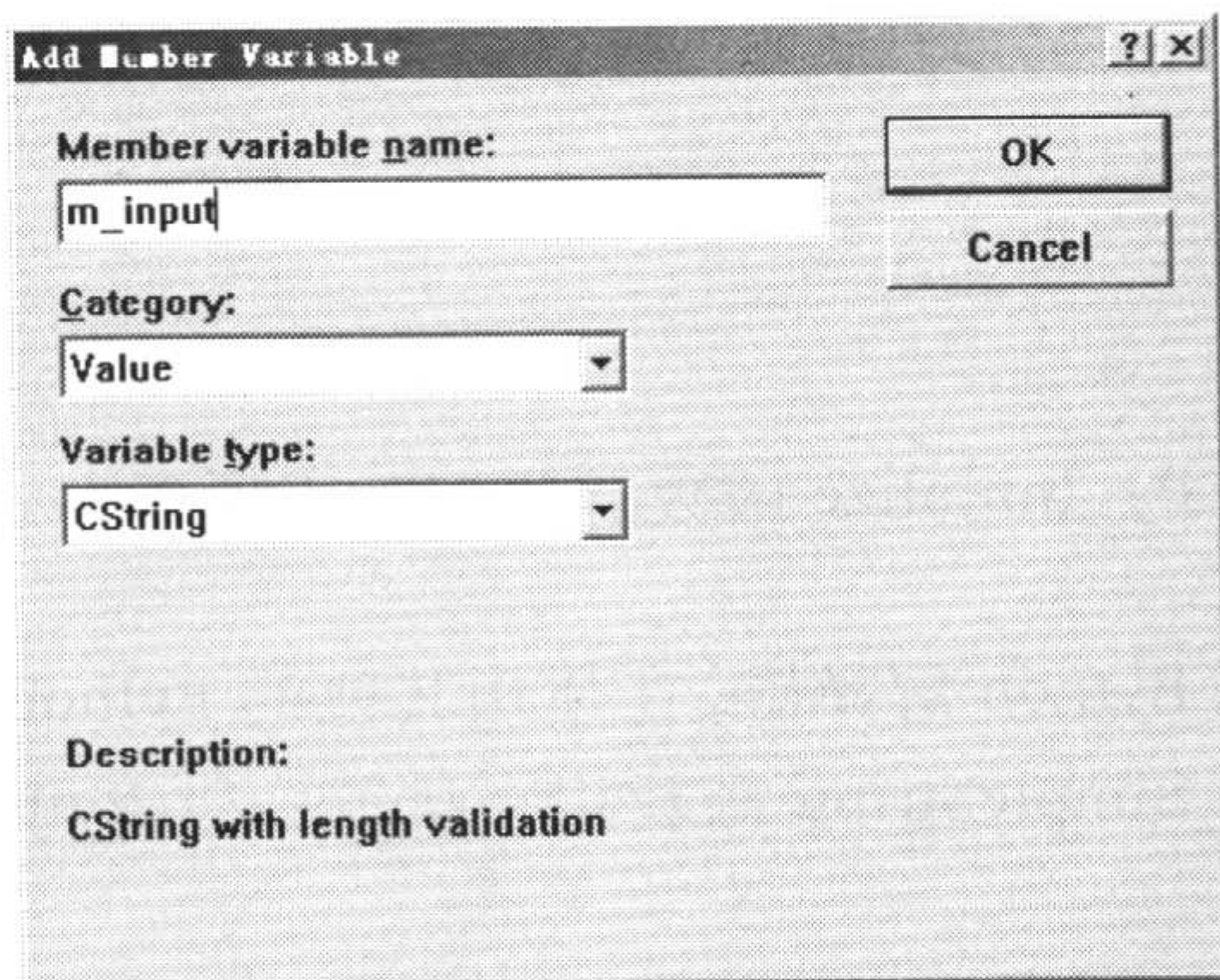


图 8-26 设置变量名

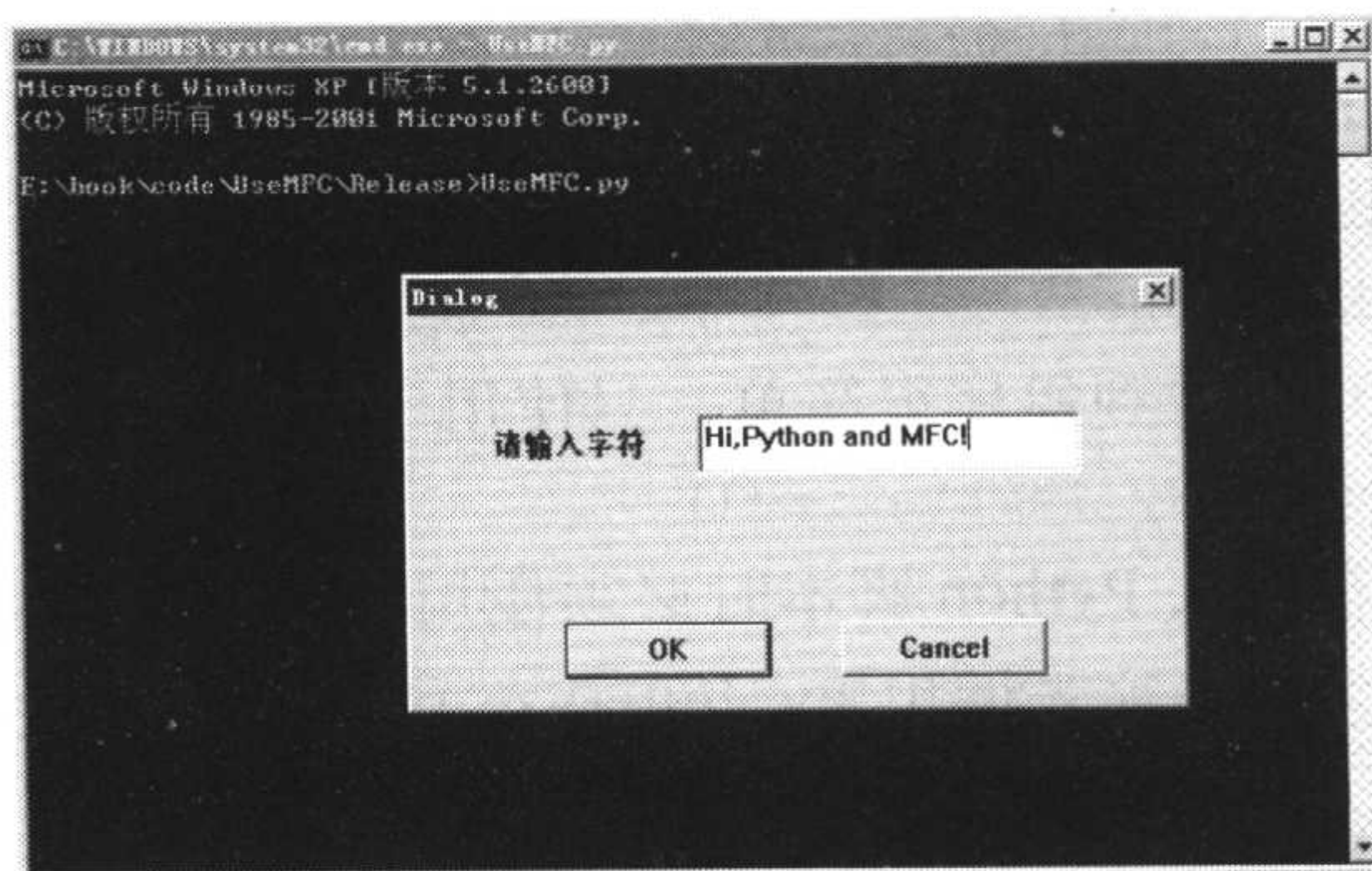


图 8-27 脚本运行弹出对话框

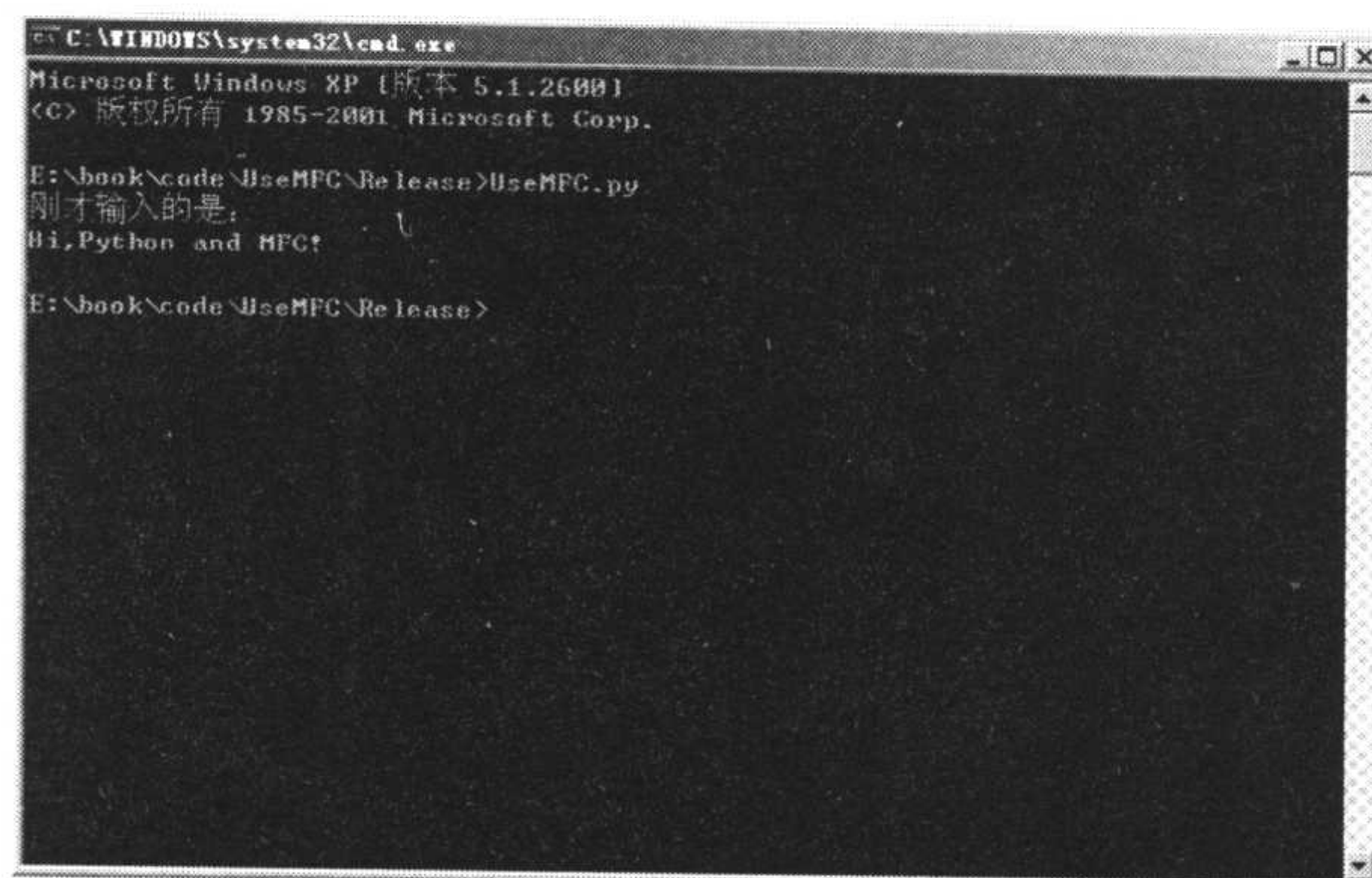


图 8-28 脚本获得文本框中的文本

8.2 在 C/C++ 中嵌入 Python

在 C/C++ 中嵌入 Python，可以使用 Python 提供的强大功能，通过嵌入 Python 可以替代动态链接库形式的接口，这样可以方便地根据需要修改脚本代码，而不用重新编译链接二进制的动态链接库。

8.2.1 高层次嵌入 Python

使用 Python/C API 可以在较高层次上嵌入 Python。所谓的高层次嵌入主要是指程序与脚本间没有交互。在 VC++ 6.0 中新建一个空“Win32 Console Application”，在工程中新建一个 C 源文件。将如下所示代码添加到其中。

```
#include <Python.h>
int main()
{
    Py_Initialize();                                /* Python 解释器初始化 */
    PyRun_SimpleString("print 'hi,python!'");        /* 运行字符串 */
    Py_Finalize();                                    /* 结束 Python 解释器，释放资源 */
    return 0;
}
```

编译工程，运行程序后输出如下所示。

```
hi,python!
```

可以看到程序很简单，只使用了 3 个函数。其中 Py_Initialize 函数的原型如下所示。

```
void Py_Initialize()
```

在嵌入 Python 脚本时必须使用该函数，它初始化 Python 解释器。在使用其他的 Python/C API 之前必须先调用 Py_Initialize 函数。其中 PyRun_SimpleString 函数用来执行一段 Python 代码。其函数原型如下所示。

```
int PyRun_SimpleString(const char *command)
```

在程序的最后使用了 Py_Finalize 函数，其原型如下所示。

```
void Py_Finalize()
```

Py_Finalize 函数用于关闭 Python 解释器，释放解释器所占用的资源。

除了使用 PyRun_SimpleString 函数以外，还可以使用 PyRun_SimpleFile() 函数来运行“.py”脚本文件。其函数原型如下所示。

```
int PyRun_SimpleFile( FILE *fp, const char *filename)
```

其参数含义如下。

- fp: 打开的文件指针。
- filename: 要运行的 Python 脚本文件名。

在 Windows 下使用该函数时需要注意所使用的编译器版本。由于官方发布的 Python 是由 Visual Studio 2003.NET 编译的。如果使用其他版本的编译器，由于版本差异导致 FILE 的定义有所区别，因此使用其他版本的编译器会导致程序崩溃。

为了简便起见可以使用如下方式来代替 PyRun_SimpleFile 函数实现同样的功能。

```
PyRun_SimpleString("execfile('file.py')"); # 使用 execfile 运行 Python 脚本文件
```

8.2.2 较低层次嵌入 Python

在上一节的例子中只使用简单的函数就完成了在 C 语言中嵌入 Python。但如果需要在 C 程序中用 Python 脚本传递参数，或者获得 Python 脚本的返回值，则要使用更多的函数来编写 C 程序。由于 Python 有自己的数据类型，因此在 C 程序中要使用专门的 API 对相应的数据类型进行操作。常用的函数有以下几种。

1. 数字与字符串处理

在 Python/C API 中提供了 Py_BuildValue() 函数对数字和字符串进行转换处理，使之变成 Python 中相应的数据类型。其函数原型如下所示。

```
PyObject* Py_BuildValue( const char *format, ...)
```

其参数含义如下。

- format: 格式化字符串，如表 8-1 所示。

Py_BuildValue() 函数中剩余的参数即要转换的 C 语言中的整型、浮点型或者字符串等。其返回值为 PyObject 型的指针。在 C 语言中，所有的 Python 类型都被声明为 PyObject 型。

2. 列表操作

在 Python/C API 中提供了 PyList_New() 函数用以创建一个新的 Python 列表。PyList_New() 函数的返回值为所创建的列表。其函数原型如下所示。

```
PyObject* PyList_New( Py_ssize_t len)
```

其参数含义如下。

- len: 所创建列表的长度。

当列表创建以后，可以使用 PyList_SetItem() 函数向列表中添加项。其函数原型如下所示。


```
int PyList_SetItem( PyObject *list, Py_ssize_t index, PyObject *item)
```

其参数含义如下。

- list: 要添加项的列表。
- index: 所添加项的位置索引。
- item: 所添加项的值。

同样可以使用 Python/C API 中 PyList_GetItem() 函数来获取列表中某项的值。

PyList_GetItem() 函数返回项的值。其函数原型如下所示。

```
PyObject* PyList_GetItem( PyObject *list, Py_ssize_t index)
```

其参数含义如下。

- list: 要进行操作的列表。
- index: 项的位置索引。

Python/C API 中提供了与 Python 中列表操作相对应的函数。例如列表的 append 方法对应于 PyList_Append() 函数。列表的 sort 方法对应于 PyList_Sort() 函数。列表的 reverse 方法对应于 PyList_Reverse() 函数。其函数原型分别如下所示。

```
int PyList_Append( PyObject *list, PyObject *item)
```

```
int PyList_Sort( PyObject *list)
```

```
int PyList_Reverse( PyObject *list)
```

对于 PyList_Append() 函数，其参数含义如下。

- list: 要进行操作的列表。
- item: 要参加的项。

对于 PyList_Sort() 和 PyList_Reverse() 函数，其参数含义相同。

- list: 要进行操作的列表。

3. 元组操作

在 Python/C API 中提供了 PyTuple_New() 函数，用以创建一个新的 Python 元组。

PyTuple_New() 函数返回所创建的元组。其函数原型如下所示。

```
PyObject* PyTuple_New( Py_ssize_t len)
```

其参数含义如下。

- len: 所创建元组的长度。

当元组创建以后，可以使用 PyTuple_SetItem() 函数向元组中添加项。其函数原型如下所示。

```
int PyTuple_SetItem( PyObject *p, Py_ssize_t pos, PyObject *o)
```

其参数含义如下所示。

- p: 所进行操作的元组。
- pos: 所添加项的位置索引。

- o: 所添加的项值。

可以使用 Python/C API 中 `PyTuple_GetItem()` 函数来获取元组中某项的值。`PyTuple_GetItem()` 函数返回项的值。其函数原型如下所示。

```
PyObject* PyTuple_GetItem( PyObject *p, Py_ssize_t pos)
```

其参数含义如下。

- p: 要要进行操作的元组。
- pos: 项的位置索引。

当元组创建以后可以使用 `_PyTuple_Resize()` 函数重新调整元组的大小。其函数原型如下所示。

```
int _PyTuple_Resize( PyObject **p, Py_ssize_t newsize)
```

其参数含义如下。

- p: 指向要要进行操作的元组的指针。
- newsize: 新元组的大小。

4. 字典操作

在 Python/C API 中提供了 `PyDict_New()` 函数用以创建一个新的字典。`PyDict_New()` 函数返回所创建的字典。其函数原型如下所示。

```
PyObject* PyDict_New()
```

当字典创建后, 可以使用 `PyDict_SetItem()` 函数和 `PyDict_SetItemString()` 函数向字典中添加项。其函数原型分别如下所示。

```
int PyDict_SetItem( PyObject *p, PyObject *key, PyObject *val)
int PyDict_SetItemString( PyObject *p, const char *key, PyObject *val)
```

其参数含义如下。

- p: 要要进行操作的字典。
- key: 添加项的关键字, 对于 `PyDict_SetItem()` 函数其为 `PyObject` 型, 对于 `PyDict_SetItemString()` 函数其为 `char` 型。
- val: 添加项的值。

使用 Python/C API 中的 `PyDict_GetItem()` 函数和 `PyDict_GetItemString()` 函数来获取字典中某项的值。它们都返回项的值。其函数原型分别如下所示。

```
PyObject* PyDict_GetItem( PyObject *p, PyObject *key)
PyObject* PyDict_GetItemString( PyObject *p, const char *key)
```

其参数含义如下。

- p: 要要进行操作的字典。
- key: 添加项的关键字, 对于 `PyDict_GetItem()` 函数其为 `PyObject` 型, 对于 `PyDict_GetItemString()` 函数其为 `char` 型。

使用 Python/C API 中的 `PyDict_DelItem()` 函数和 `PyDict_DelItemString()` 函数可以删除字

典中的某一项。其函数原型如下所示。

```
int PyDict_DelItem( PyObject *p, PyObject *key)
int PyDict_DelItemString( PyObject *p, char *key)
```

其参数含义如下。

- p: 要进行操作的字典。
- key: 添加项的关键字, 对于 PyDict_DelItem() 函数其为 PyObject 型, 对于 PyDict_DelItemString() 函数其为 char 型。

使用 Python/C API 中的 PyDict_Next() 函数可以对字典进行遍历。其函数原型如下所示。

```
int PyDict_Next( PyObject *p, Py_ssize_t *ppos, PyObject **pkey, PyObject **pvalue)
```

其参数含义如下。

- p: 要进行遍历的字典。
- ppos: 字典中项的位置, 应该被初始化为 0。
- pkey: 返回字典的关键字。
- pvalue: 返回字典的值。

在 Python/C API 中提供了与 Python 中字典操作相对应的函数。例如字典的 item 方法对应于 PyDict_Items() 函数。字典的 keys 方法对应于 PyDict_Keys() 函数。字典的 values 方法对应于 PyDict_Values() 函数。其函数原型分别如下所示。

```
PyObject* PyDict_Items( PyObject *p)
PyObject* PyDict_Keys( PyObject *p)
PyObject* PyDict_Values( PyObject *p)
```

其参数含义如下。

- p: 要进行操作的字典。

5. 释放资源

Python 使用引用计数机制对内存进行管理, 实现自动垃圾回收。在 C/C++ 中使用 Python 对象时, 应正确地处理引用计数, 否则容易导致内存泄漏。在 Python/C API 中提供了 Py_CLEAR()、Py_DECREF() 等宏来对引用计数进行操作。

当使用 Python/C API 中的函数创建列表、元组、字典等后, 就在内存中生成了这些对象的引用计数。在对其完成操作后应该使用 Py_CLEAR()、Py_DECREF() 等宏来销毁这些对象。其原型分别如下所示。

```
void Py_CLEAR( PyObject *o)
void Py_DECREF( PyObject *o)
```

其参数含义如下。

- o: 要进行操作的对象。

对于 Py_CLEAR() 其参数可以为 NULL 指针, 此时, Py_CLEAR() 不进行任何操作。而对于 Py_DECREF() 其参数不能为 NULL 指针, 否则将导致错误。

6. 模块与函数

使用 Python/C API 中的 `PyImport_Import()` 函数可以在 C 程序中导入 Python 模块。
`PyImport_Import()` 函数返回一个模块对象。其函数原型如下所示。

```
PyObject* PyImport_Import( PyObject *name)
```

其参数含义如下。

- `name`: 要导入的模块名。

使用 Python/C API 中的 `PyObject_CallObject()` 函数和 `PyObject_CallFunction()` 函数，可以在 C 程序中调用 Python 中的函数。其参数原型分别如下所示。

```
PyObject* PyObject_CallObject( PyObject *callable_object, PyObject *args)
```

```
PyObject* PyObject_CallFunction( PyObject *callable, char *format, ...)
```

对于 `PyObject_CallObject()` 函数，其参数含义如下。

- `callable_object`: 要调用的函数对象。
- `args`: 元组形式的参数列表。

对于 `PyObject_CallFunction()` 函数，其参数含义如下。

- `callable_object`: 要调用的函数对象。
- `format`: 指定参数的类型。
- `...`: 向函数传递的参数。

使用 Python/C API 中的 `PyModule_GetDict()` 函数可以获得 Python 模块中的函数列表。
`PyModule_GetDict()` 函数返回一个字典。字典中的关键字为函数名，值为函数的调用地址。
其函数原型如下所示。

```
PyObject* PyModule_GetDict( PyObject *module)
```

其参数含义如下。

- `module`: 已导入的模块对象。

8.2.3 在 C 中嵌入 Python 实例

在 VC++ 6.0 中新建一个名为 “EmbPython” 的空 “Win32 Console Application” 工程。向其添加如下所示的 “EmbPython.c” 文件。

```
#include <stdio.h>
#include <Python.h>
int main(int argc, char* argv[])
{
    PyObject *modulename, *module, *dic, *func, *args, *rel, *list;
    char *funcname1 = "sum";
    char *funcname2 = "strsplit";
    int i;
    Py_ssize_t s;
    printf("---在 C 中嵌入 Python---\n");
```

```

/* Python 解释器的初始化*/
Py_Initialize();
if(!Py_IsInitialized())
{
    printf("初始化失败!");
    return -1;
}
/* 导入 Python 模块,并检验是否正确导入 */
modulename = Py_BuildValue("s", "pytest");
module = PyImport_Import(modulename);
if(!module)
{
    printf("导入 pytest 失败!");
    return -1;
}
/* 获得模块中函数并检验其有效性 */
dic = PyModule_GetDict(module);
if(!dic)
{
    printf("错误!\n");
    return -1;
}
/* 获得 sum 函数地址并验证 */
func = PyDict_GetItemString(dic, funcname1);
if(!PyCallable_Check(func))
{
    printf("不能找到函数 %s", funcname1);
    return -1;
}
/* 构建列表 */
list = PyList_New(5);
printf("使用 Python 中的 sum 函数求解下列数之和\n");
for (i = 0; i < 5; i++)
{
    printf("%d\t", i);
    PyList_SetItem(list, i, Py_BuildValue("i", i));
}
printf("\n");
/* 构建 sum 函数的参数元组*/
args = PyTuple_New(1);
PyTuple_SetItem(args, 0, list);
/* 调用 sum 函数 */
PyObject_CallObject(func, args);
/* 获得 strsplit 函数地址并验证*/
func = PyDict_GetItemString(dic, funcname2);
if(!PyCallable_Check(func))
{
    printf("不能找到函数 %s", funcname2);
    return -1;
}

```

```

}
/* 构建 strsplit 函数的参数元组 */
args = PyTuple_New(2);
printf("使用 Python 中的函数分割以下字符串:\n");
printf("this is an example\n");
PyTuple_SetItem(args, 0, Py_BuildValue("s", "this is an example"));
PyTuple_SetItem(args, 1, Py_BuildValue("s", " "));
/* 调用 strsplit 函数并获得返回值 */
rel = PyObject_CallObject(func, args);
s = PyList_Size(rel);
printf("结果如下所示:\n");
for ( i = 0; i < s; i ++ )
{
    printf("%s\n", PyString_AsString(PyList_GetItem(rel, i)));
}
/* 释放资源 */
Py_DECREF(list);
Py_DECREF(args);
Py_DECREF(module);
/* 结束 Python 解释器 */
Py_Finalize();
printf("按回车键退出程序:\n");
getchar();
return 0;
}

```

程序输出如下所示。

==在 C 中嵌入 Python==

使用 Python 中的 sum 函数求解下列数之和

0 1 2 3 4

Using Function sum

The result is: 10

使用 Python 中的函数分割以下字符串:

this is an example

结果如下所示:

this

is

an

example

按回车键退出程序:

8.3 语言的黏合剂 SWIG

SWIG 可以将 C/C++ 编写的程序以扩展的形式链接到各种高级编程语言中。SWIG 支持的语言非常广泛,几乎支持目前所有流行的语言。使用 SWIG 可以轻易地使用 C/C++ 为 Python 编写扩展。

8.3.1 在 Windows 集成开发环境中使用 SWIG

由于在 Windows 平台下一般使用集成开发环境来创建编译工程。为了在集成环境中使用 SWIG，需要对其进行设置。以下步骤以 VC++ 6.0 为例将 SWIG 集成到 VC++ 6.0 中。

(1) 单击 **【Tools】 | 【Options】** 命令，弹出如图 8-29 所示的对话框。

(2) 单击 **【Directories】** 标签，选择 **【Show directories for】** 下拉列表框中的 **【Executable files】** 项，将 SWIG 所在的目录添加到 Directories 列表中，如图 8-30 所示。

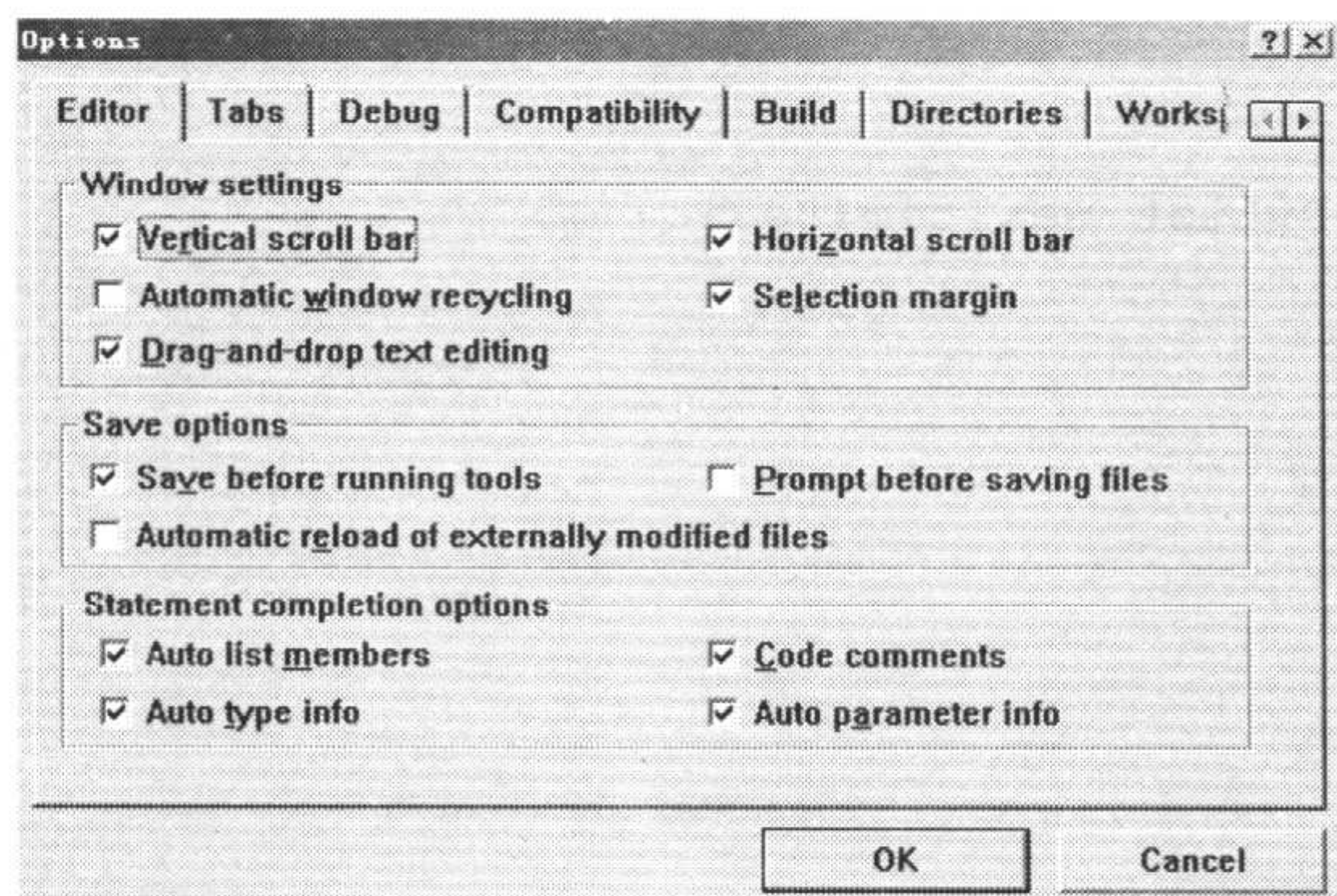


图 8-29 Options 对话框

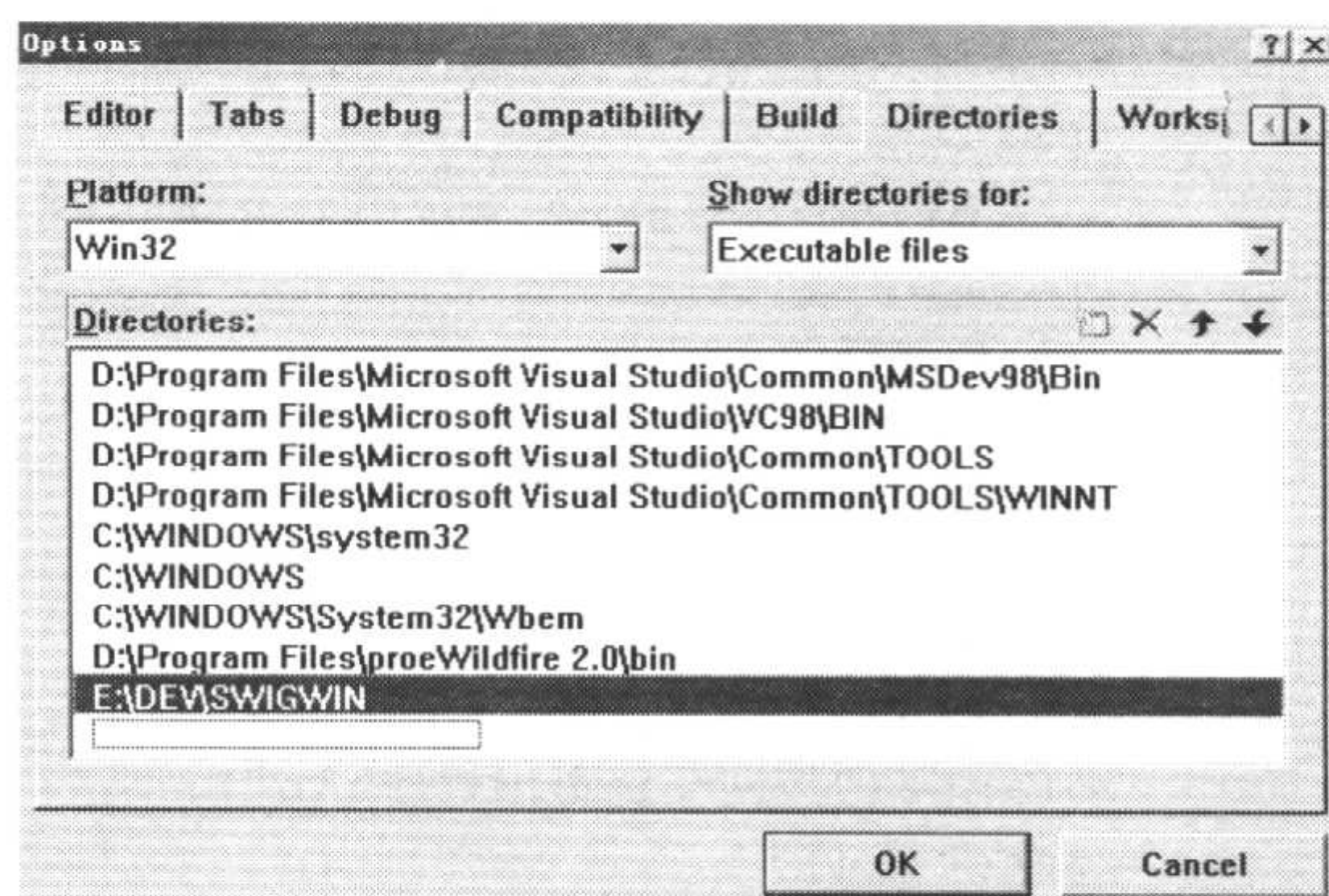


图 8-30 将 SWIG 添加到可执行文件路径列表

(3) 单击 **【OK】** 按钮，完成操作。

(4) 新建一个名为“useSWIG”的空“Win32 Dynamic_Link Library”工程。向其中添加 3 个文件“useSWIG.i”，“useSWIG.c”，“useSWIG_wrap.c”。其中“useSWIG.i”为 SWIG 的接口文件，“useSWIG.c”为编写 Python 扩展的 C 源文件，“useSWIG_wrap.c”为 SWIG 生成的接口文件。

(5) 向“useSWIG.i”中添加如下内容。

```
%module useSWIG
%inline %{
extern void showSWIG();
%}
```

向“useSWIG.c”中添加如下内容。

```
#include <stdio.h>
void showSWIG()
{
    printf("Hi, SWIG!\n");
}
```

(6) 单击 **【Project】 | 【Settings】** 命令，弹出工程设置对话框。选择 **【Settings For】** 下拉列表框中的 **【Win32 Release】** 项。选中左侧树形列表中的“useSWIG.i”文件，单击 **【Custom Build】** 标签。在 **【Description】** 文本框中输入“SWIG”。在 **【Commands】** 文本框中输入“swig

-python -o \$(ProjDir)\\$(InputName)_wrap.c \$(InputPath)”，即使用 SWIG 对 “useSWIG.i” 进行编译。在【Outputs】文本框中输入 “\$(ProjDir)\\$(InputName)_wrap.c”，如图 8-31 所示。

(7) 选中左侧树形列表中的整个工程，单击【Link】标签，将【Output file name】文本框中的 “Release/useSWIG.dll” 改为 “_useSWIG.pyd”，如图 8-32 所示。这是 SWIG 的规定，因为 SWIG 将自动生成一个 Python 脚本文件来调用所编写的 Python 扩展。因此在使用 Python 扩展时应该导入 SWIG 所生成的 Python 脚本，而不是直接导入生成的 Python 扩展文件。

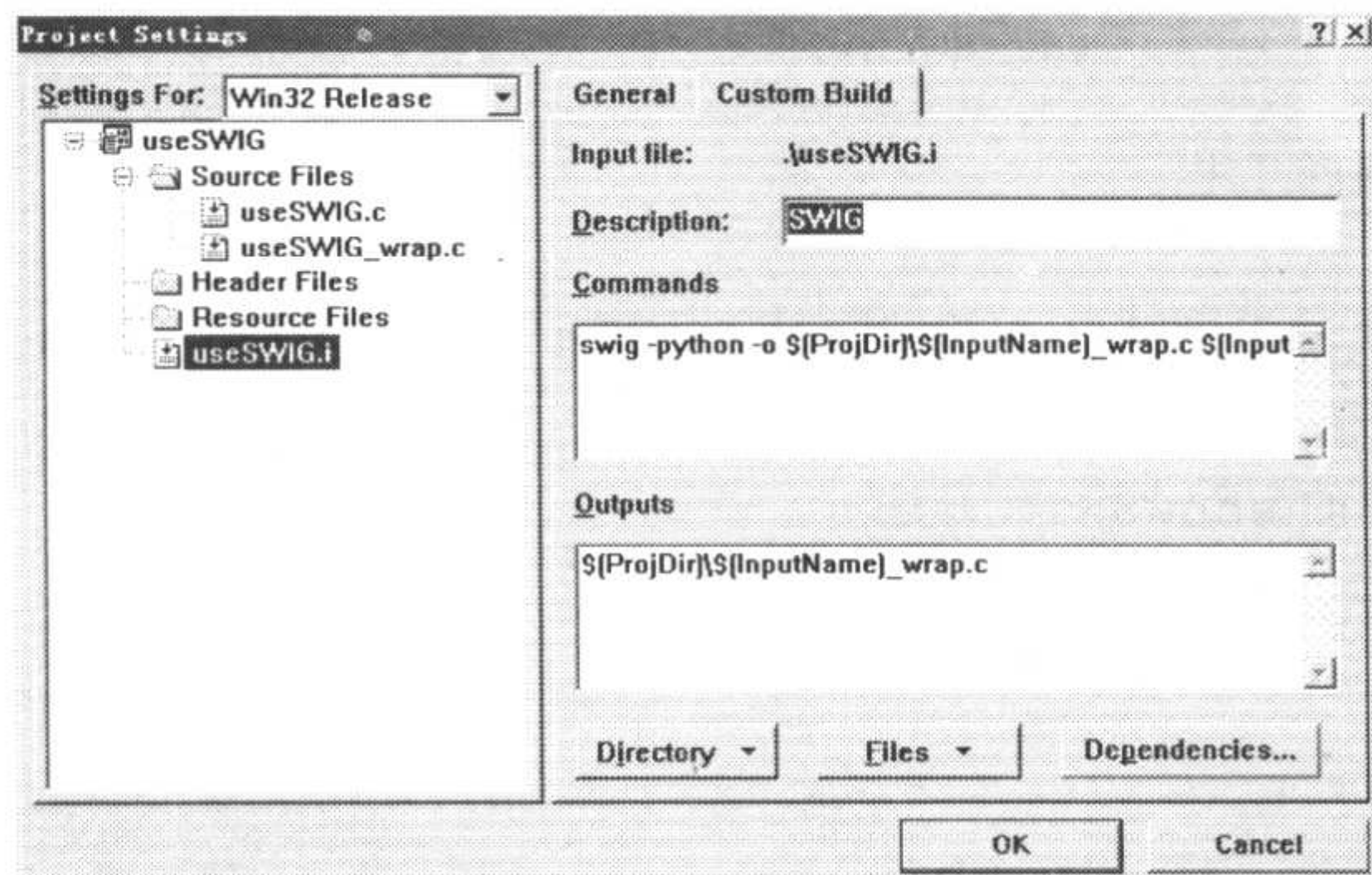


图 8-31 设置 SWIG 接口文件的编译方式

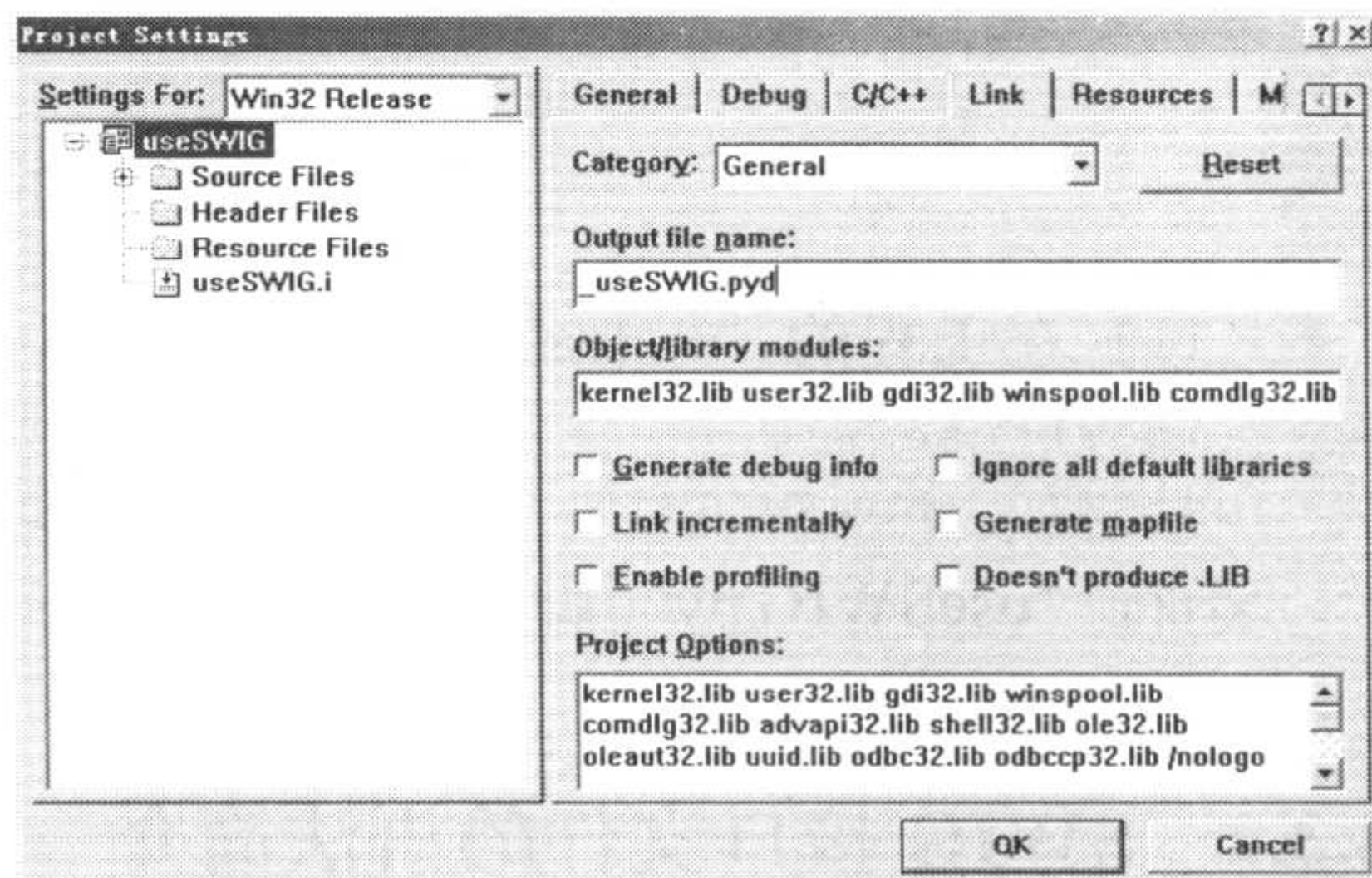


图 8-32 设置 Link 选项

(8) 单击【C/C++】标签，选择【Category】下拉列表框中的【Code Generation】项，选择【Use run-time library】下拉列表框中的【Multithreaded DLL】项，如图 8-33 所示。选择【Category】下拉列表框中的【Preprocessor】项，在【Preprocessor definitions】文本框中添加 “_WIN32_”，如图 8-34 所示。

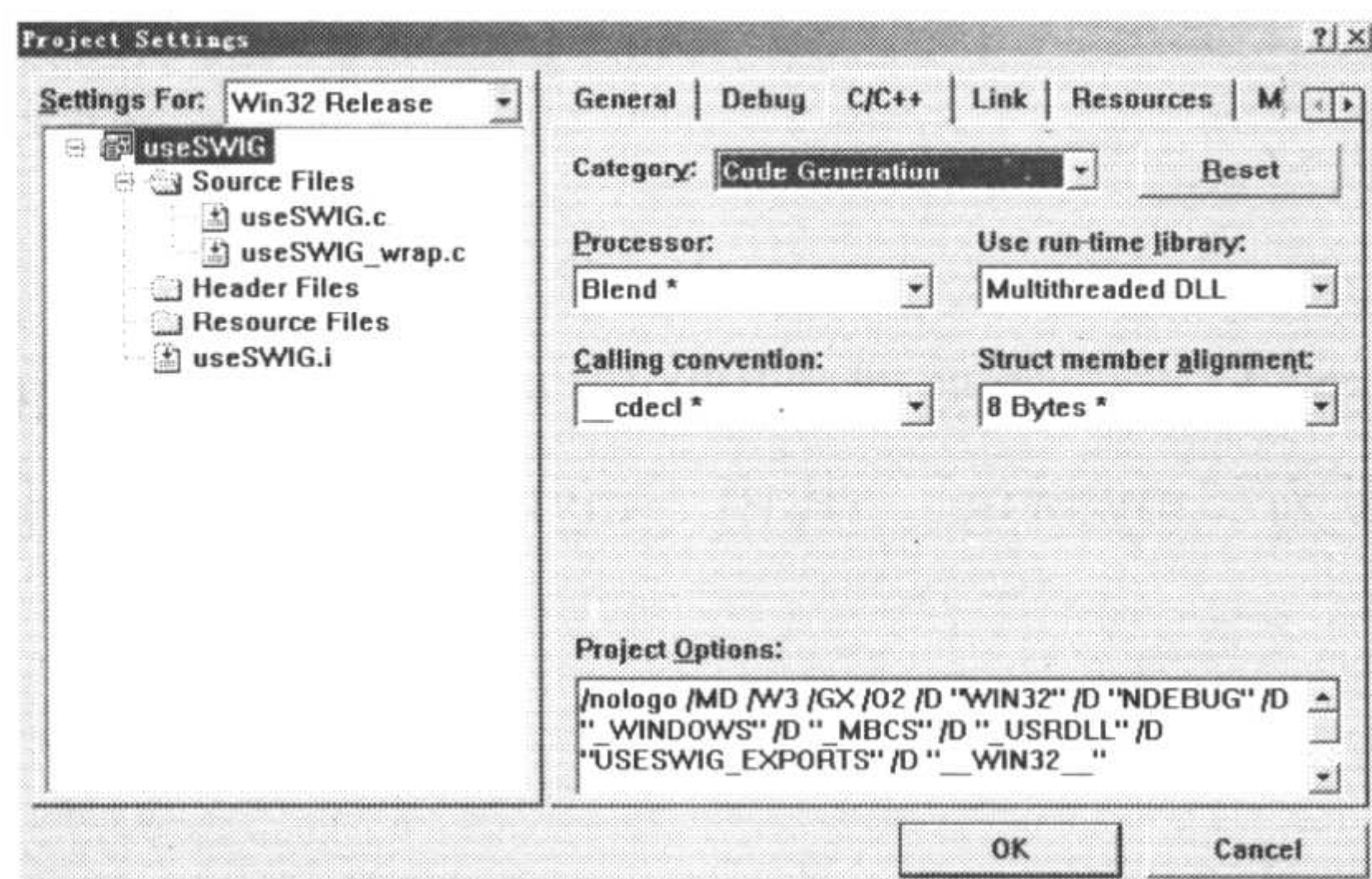


图 8-33 设置 C/C++选项

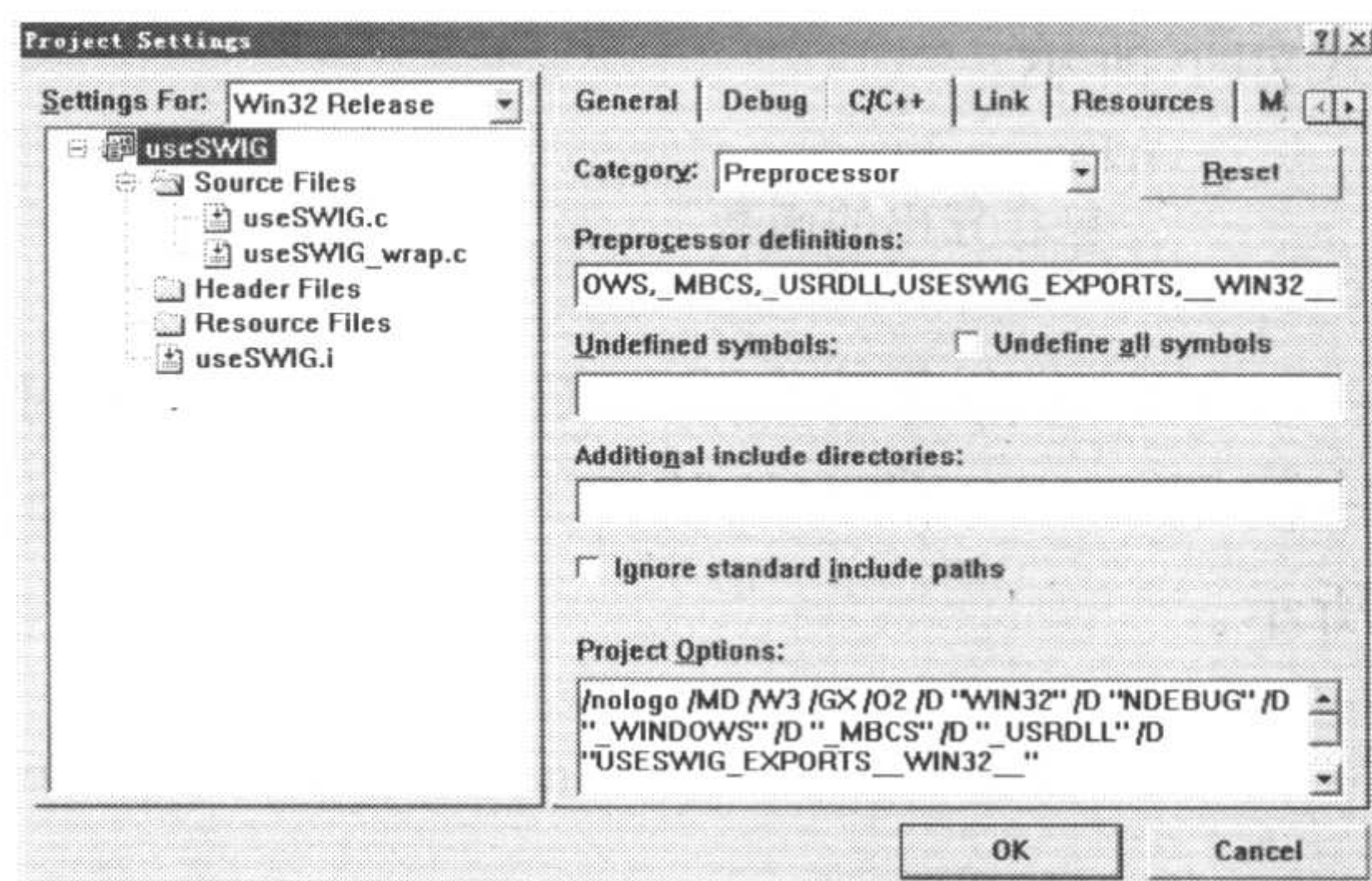


图 8-34 添加预处理定义

(9) 单击【OK】按钮完成工程设置。

(10) 单击【Build】|【Batch Build】命令，弹出如图 8-35 所示的对话框，将【useSWIG-Win32 Debug】单选框前的勾去掉。单击【Build】按钮生成 Python 扩展。

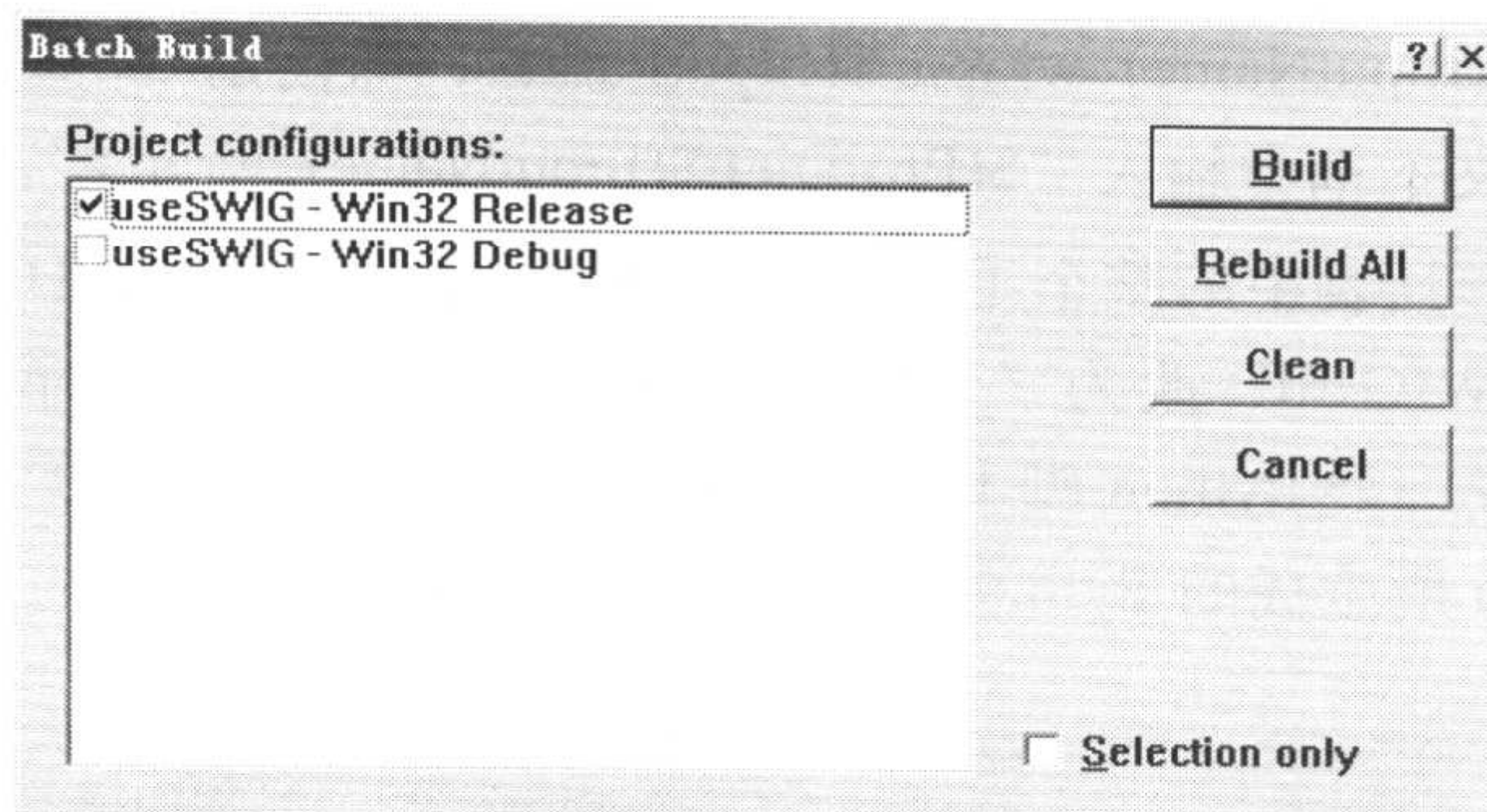


图 8-35 编译工程

(11) 在工程的“Release”目录下创建“use.py”脚本，向其中添加如下所示内容。

```
import useSWIG                # 此处导入的是 SWIG 生成的 Python 脚本
useSWIG.showSWIG()           # 调用扩展中的 showSWIG 方法
```

运行“useSWIG.py”脚本后，输出如下所示。

```
Hi, SWIG!
```

8.3.2 SWIG 接口文件的语法简介

由上一节中的例子可以看出，使用 SWIG 编写 Python 扩展要简单得多。无需在 C 源文件中使用 Python/C API，只要编写相应的 SWIG 接口文件即可。使用 Python/C API 编写 Python 扩展时需要定义导出函数，并初始化模块名。相应地在 SWIG 接口文件中可以使用 %module 设置 Python 扩展的模块名。如下所示代码定义一个名为 example 的模块名。

```
%module example
```

使用 PyMethodDef 定义的方法列表，相应地在 SWIG 接口文件中可以使用 %inline。其形式如下所示。

```
%inline %{
    包含导出的函数
%}
```

SWIG 将自动生成与 C 中参数类型相对应的 Python 中的类型。按照上一节中在 VC++ 6.0 中使用 SWIG 的方法将 8.1 节中的“myext”改写为“myswig”。其中“myswig.c”内容如下所示。

```
#include <windows.h>
int show(char *message, char *title)
{
    int r;
    r = MessageBox(NULL, message, title, MB_OK);
    return r;
}
```

其中“myswig.i”内容如下所示。

```
%module myswig
```



```
%inline %{  
extern int show(char *message, char *title);  
%}
```

编写使用“myswig”的“use.py”脚本，其内容如下所示。

```
import myswig  
myswig.show('Hi, SWIG', 'MYSWIG')
```

脚本运行后如图 8-36 所示。



图 8-36 使用 SWIG

使用 SWIG 编写 Python 扩展可以省去很多代码，而这些代码都将由 SWIG 自动生成。使用 SWIG 不必考虑 C 与 Python 之间数据类型转换的问题。

8.4 混合系统接口 Boost.Python

Boost 是一个可移植的 C++ 标准库，相当于 STL 的延续和扩充。Boost 库也为 C++ 编写 Python 扩展提供了支持。如果使用 C++ 为 Python 编写扩展，使用 Boost.Python 将使程序变得简单。

8.4.1 编译 Boost.Python

首先要编译 Boost.Python 才可以使用其进行编程。由于 Boost 库过于庞大，如果没有其他需要，则可以仅对 Boost.Python 进行编译。以 VC++ 6.0 为例，其编译步骤如下所示。

- (1) 从 Boost 官方网站下载 Boost 库源文件，将其解压至某一目录中。
- (2) 将 Boost 解压至的目录添加到 VC++ 6.0 的【Include files】中，如图 8-37 所示。
- (3) 进入 Boost 目录下的“libs\python\build\VisualStudio”子目录中，在 VC++ 6.0 中打开其中的“boost_python.dsw”文件。
- (4) 单击【Build】|【Batch Build】命令，分别编译 Boost.Python 的 Debug 和 Release 版。
- (5) 编译完成后将在 Boost 目录下的“libs\python\build\bin-stage”子目录中生成动态链接库和库文件。由于使用 Boost.Python 编写的 Python 扩展在运行时根据版本不同需要“boost_python.dll”和“boost_python_debug.dll”文件。为了方便，可以将其放到 Windows 安装目录下的“system32”目录下。否则，需要将其和 Python 扩展放在同一目录中。
- (6) 将 Boost 目录下的“libs\python\build\bin-stage”子目录添加到 VC++ 6.0 的【Library

files】中，如图 8-38 所示。

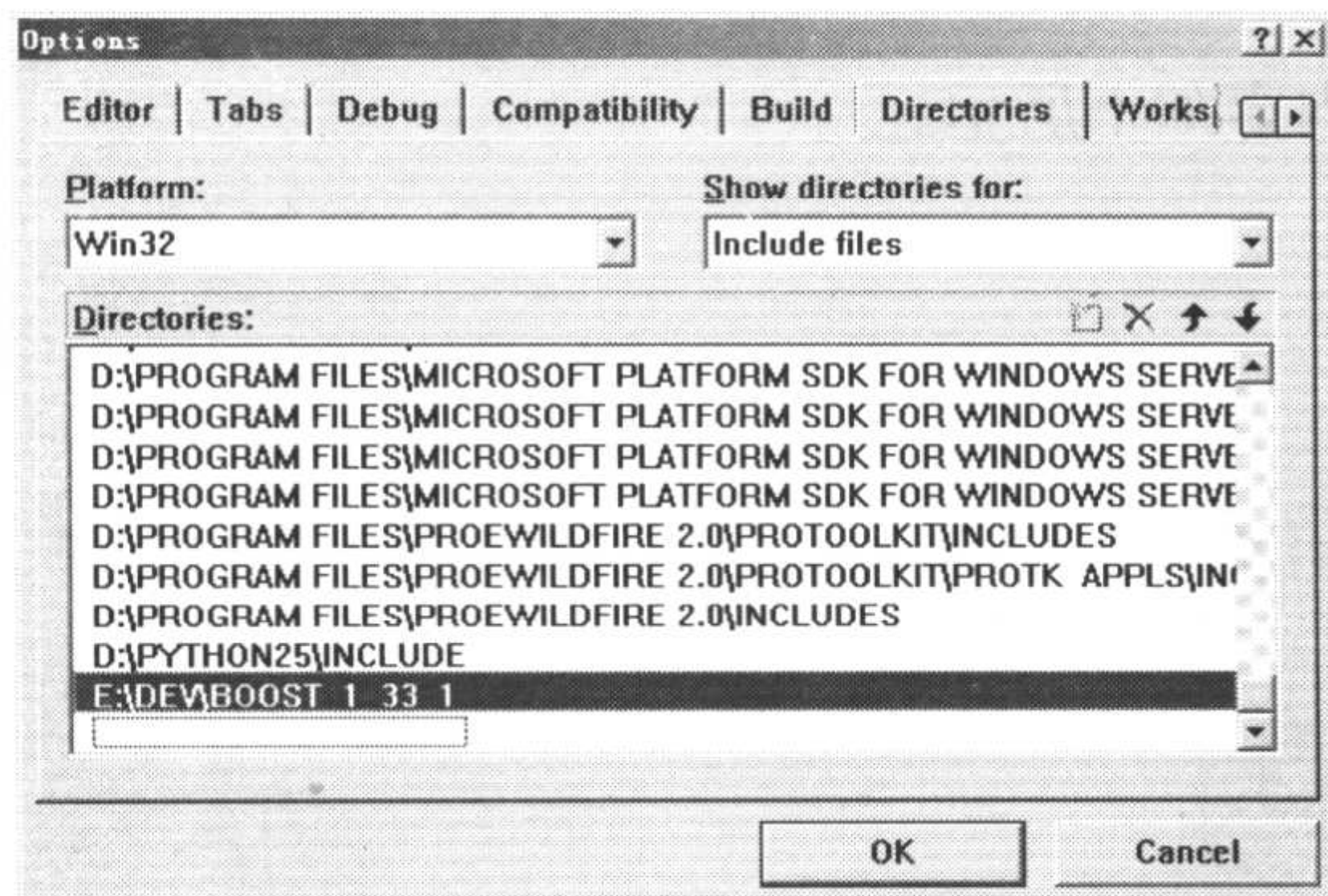


图 8-37 VC++ 6.0 头文件

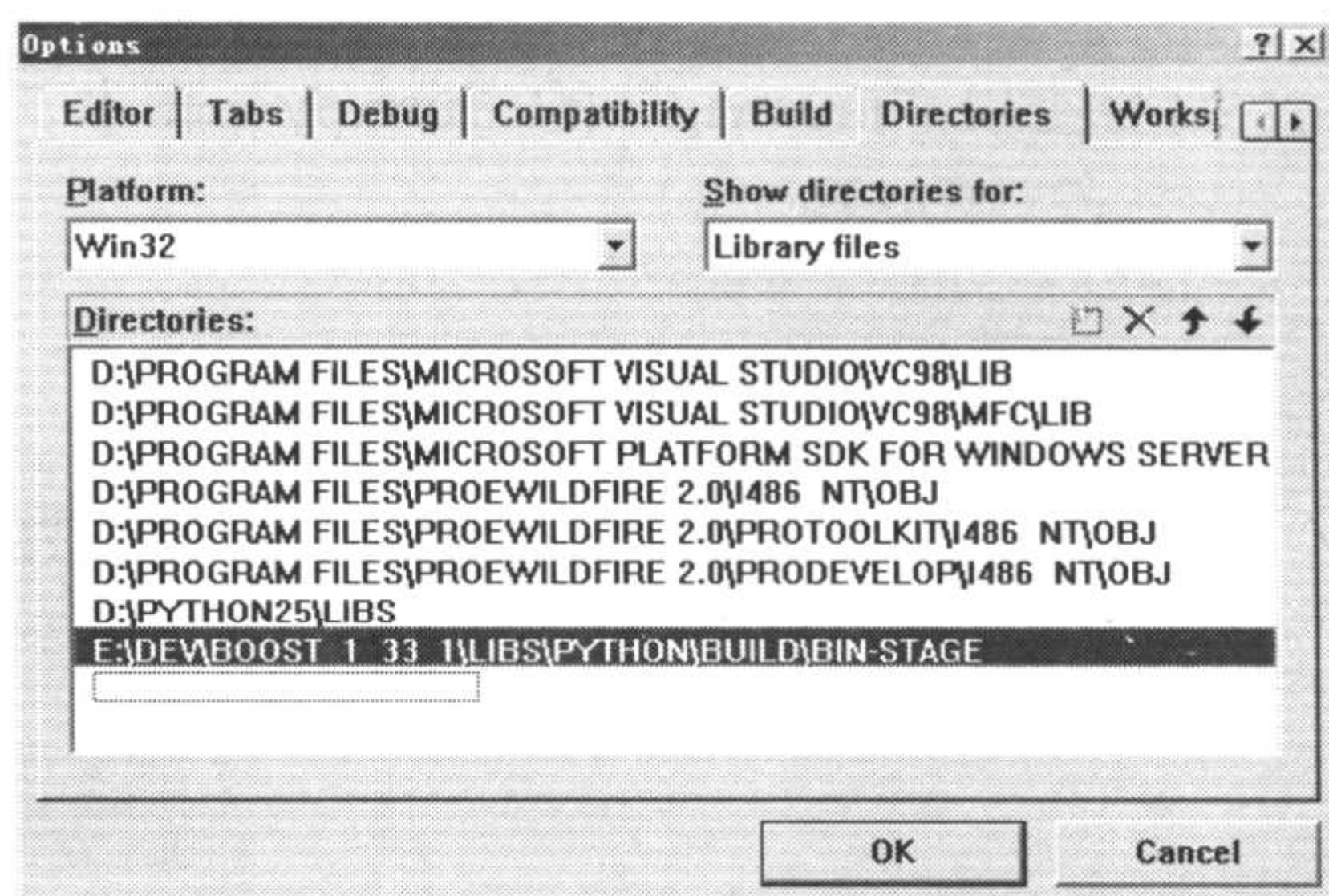


图 8-38 VC++ 6.0 库文件

完成上述设置后就可以使用 Boost.Python 编写 Python 扩展了。

8.4.2 使用 Boost.Python 扩展和嵌入 Python

通过 Boost.Python 可以在 Python 内使用 C++类和函数。和 SWIG 一样 Boost.Python 简化了编写 Python 扩展的代码，而不用使用 Python/C API。但与 SWIG 不同，Boost.Python 是一个类库，无需再使用接口文件。

1. 初始化和方法列表

在 Boost.Python 中可以通过使用 BOOST_PYTHON_MODULE 来命名模块名。在 BOOST_PYTHON_MODULE 中则可以使用 def 来实现使用 Python/C API 定义的方法列表。以下是一个简单的例子。

```
void show() // 声明 show 函数
{
    cout << "Boost.Python";
}
BOOST_PYTHON_MODULE(example) // 使用 BOOST_PYTHON_MODULE 命名模块名为“example”
{
    def("show", show); // 相当于定义方法列表
}
```

2. 导出类

通过 Boost.Python 可以将将在 C++中定义类及其方法、属性等导入 Python 中。以下程序在 C++中定义一个 Message 类，然后通过 BOOST_PYTHON_MODULE 将其导入 Python 中。

```
#include <string.h>
#include <iostream.h>
#include <boost/python.hpp>
using namespace boost::python;
#pragma comment(lib, "boost_python.lib")
```

```

class Message
{
public:
    std::string msg;
    Message(std::string m)
    {
        msg = m;
    }
    void set(std::string m)
    {
        msg = m;
    }
    std::string get()
    {
        return msg;
    }
};

BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message", init<std::string>())
        .def("set", &Message::set)
        .def("get", &Message::get)
        ;
}

```

以下代码在 Python 中使用编译好的 Message 模块。

```

>>> import Message
>>> c = Message.Message('hi')
>>> c.get()
'hi'
>>> c.set('Boost.Python')
>>> c.get()
'Boost.Python'

```

3. 类的成员属性

在 C++ 中类的成员可以使用关键字声明为不同的属性。而在 Python 中则依靠类属性的命名方式。使用 Boost.Python 可以将其 C++ 中类成员的属性传递给 Python。如下所示代码使用 Boost.Python 来处理类成员的属性。将 BOOST_PYTHON_MODULE 中的代码改为如下所示。

```

BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message", init<std::string>())
        .def("set", &Message::set)
        .def("get", &Message::get)
        .def_readwrite("msg", &Message::msg);
    ;
}

```

此处将 Message 类中的成员 msg 设置为可读写，还可以使用 “.def_readonly” 将其设置

为只读属性。对于类中的私有成员还可以使用“.add_property”将其操作函数设置为 Python 类中的属性。如下代码使用“.add_property”对私有成员进行操作。

```
BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message", init<std::string>())
        .add_property("msg", &Message::get, &Message::set);
}
```

以下代码在 Python 中使用编译好的 Message 模块。

```
>>> import Message
>>> s = Message.Message('hi')
>>> s.msg
'hi'
>>> s.msg = 'boot.'
>>> s.msg
'boot'
```

4. 类的继承

C++中类的继承关系也可以通过 Boost.Python 反映到 Python 模块中。以下代码将父类和子类分别导入到 Python 模块中。

```
#include <string.h>
#include <iostream.h>
#include <boost/python.hpp>
using namespace boost::python;
#pragma comment(lib, "boost_python.lib")
class Message
{
public:
    std::string msg;
    Message(std::string m)
    {
        msg = m;
    }
    void set(std::string m)
    {
        msg = m;
    }
    std::string get()
    {
        return msg;
    }
};
class Msg:public Message
{
public:
    int count;
    Msg(std::string m):Message(m)
    {
```

```

    }
    void setcount(int n)
    {
        count = n;
    }
    int getcount()
    {
        return count;
    }
};
BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message",init<std::string>())
        .add_property("msg",&Message::get,&Message::set);
    class_<Msg, bases<Message> >("Msg",init<std::string>())
        .def("setcount", &Msg::setcount)
        .def("getcount", &Msg::getcount);
}

```

5. 运算符重载

在 Python 中运算符重载实际是类专有方法的重载。在 C++中对运算符重载后，通过 Boost.Python 可以传递给 Python。如下代码将 Msg 类的“+”运算符重载，然后通过“.def(self + self)”传递给 Python。

```

class Msg:public Message
{
public:
    int count;
    Msg(std::string m):Message(m)
    {
    }
    void setcount(int n)
    {
        count = n;
    }
    int getcount()
    {
        return count;
    }
    int operator+ (Msg x) const
    {
        int r;
        r = count + x.count;
        return r;
    }
};
BOOST_PYTHON_MODULE(Message)
{
    class_<Message>("Message",init<std::string>())
        .add_property("msg",&Message::get,&Message::set);
}

```

```
class_<Msg, bases<Message> >("Msg",init<std::string>())
    .def("setcount", &Msg::setcount)
    .def("getcount", &Msg::getcount)
    .def(self + self);
}
```

对于其他的运算符重载也可以使用同样的方法，如下所示。

```
.def(self - self)           // 相当于_sub_方法
.def(self * self)          // 相当于_mul_方法
.def(self /self)           // 相当于_div_方法
.def(self < self);         // 相当于_lt_方法
```

6. 使用 Boost.Python 在 C++ 中嵌入 Python

在 C++ 中嵌入 Python 的主要问题是 C++ 与 Python 之间的数据类型的转换。Boost.Python 对 Python/C API 进行封装，可以使用类似于 Python 的方式对数据进行声明、操作等。在 Boost.Python 中使用 Object 类封装了 Python/C API 中的 PyObject*。Boost.Python 定义了与 Python 中列表、字典等对应的类，如表 8-2 所示。

表 8-2 Boost.Python 与 Python 中部分类的对应关系

Boost.Python	Python
list	列表
dict	字典
tuple	元组
str	字符串

Boost.Python 提供了 Python 中对象的操作方法。如 str 类具有 lower、split、title、upper 等方法。而 list 类则具有 append、count、extend 等方法。以下代码使用 Boost.Python 在 C++ 中嵌入 Python。

8.4.3 使用 Pyste 代码生成器

Pyste 是 Boost.Python 自带的代码生成器。Pyste 与 SWIG 类似，对于源文件可以按照 C++ 的形式来写，只要编写相应的接口文件即可生成相应代码。Pyste 需要先安装才能使用。进入 Boost 的安装目录，然后进入 “/libs/python/pyste/install” 目录，运行 python setup.py install，完成 Pyste 安装。

由于 Pyste 需要 GCC-XML 的支持，因此需要到 GCC-XML 的官方网站 <http://www.gccxml.org> 下载 Windows 版本的 GCC-XML。安装完 GCC-XML 后，需要将其安装路径添加到系统 PATH 变量中。另外 Pyste 还需要 ElementTree 的支持，因此需要到其官方网站 <http://effbot.org> 下载安装。

编写如下所示头文件 “Num.h”。

```
class Num
```



```
{
    int value;
    void set( int n )
    {
        value = n;
    }
    int get()
    {
        return value;
    }
};
```

编写如下所示接口文件“world.pyste”。

```
Class("Num", "Num.h")
```

由于在 Windows 下文件路径的问题，使用 Pyste 时最好将其放到“Num.h”和“world.pyste”所在的目录。在 Windows 命令行中进入其目录，运行如下命令。

```
python pyste.py --module=num world.pyste
```

运行命令后将生成“num.cpp”文件，其内容如下所示。

```
// Boost Includes =====
#include <boost/python.hpp>
#include <boost/cstdint.hpp>

// Includes =====
#include <Num.h>

// Using =====
using namespace boost::python;

// Module =====
BOOST_PYTHON_MODULE(num)
{
    class_< Num >("Num", init< >())
        .def(init< const Num& >())
        ;
}
```

8.5 连接 Python 与 C 的桥梁——Pyrex

使用 Pyrex 将从 Pyrex 代码生成一个 C 源文件，编译该 C 源文件即可生成 Python 的扩展模块。从而使编写 Python 的 C 扩展更加容易。Pyrex 在 Python 中添加了类型声明，使得 Python 与 C 数据类型相互转换变得容易。

8.5.1 安装使用 Pyrex

Pyrex 的安装十分方便，从其官方网站 <http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex>

下载 Pyrex 的 ZIP 文件。将其解压后，进入其目录，运行如下所示命令将安装 Pyrex。

```
python setup.py install
```

Pyrex 安装好后有多种方法可以使用其编译 Pyrex 文件，但由于在 Windows 平台下，最新的 Python 2.5 版是使用 Visual Studio 2003.NET 编译的，如果没有安装 Visual Studio 2003.NET，则最好采用 Pyrex 的 ZIP 文件中的“pyrex.py”首先将 Pyrex 文件编译成 C 源文件，然后再将 C 源文件手工编译成 Python 扩展。首先编写如下所示“test.pyx”文件。

```
def test():  
    return 'A test file!'
```

将“pyrex.py”复制到“test.pyx”所在的目录，运行如下所示的命令。

```
pyrex.py test.pyx
```

运行完命令后将在目录中生成“test.c”，可以在 VC++ 6.0 中将“test.c”编译成 Python 扩展模块。在 Python 中调用编译好的 test 模块，如下所示。

```
>>> import test  
>>> test.test()  
'A test file!'
```

8.5.2 Pyrex 文件语法

Pyrex 文件完全使用了 Python 的语法。在编写 Pyrex 文件时，只需按照 Python 的语法编写即可。在 Pyrex 中可以定义 C 语言中的结构体、枚举等类型，也可以使用 C 语言中的数据类型。

1. 数据类型

Pyrex 定义了与 C 语言中相对应的数据类型，使用“cdef”可以在 Pyrex 中使用 C 语言中的数据类型。代码如下所示。

```
cdef char *s           # 相当于 C 中的字符串  
cdef int x,y           # 相当于 C 中的 int 型  
cdef int z[4]          # 相当于 C 中的数组  
cdef float f           # 相当于 C 中的浮点型  
cdef int *p            # 相当于 C 中的指针
```

2. 结构体、联合及枚举

除了定义 C 中的数据类型以外，还可以使用“cdef”定义 C 中的结构体、联合、枚举等。代码如下所示。

```
cdef struct Human:      # 定义结构体  
    char *name  
    char *sex  
    int age  
cdef union Fruit:       # 定义联合体  
    char *apple  
    char *banana  
    char *orange  
cdef enum:              # 定义枚举  
    Monday
```

```
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

3. 函数

使用“cdef”可以像使用 Python 中的“def”一样来定义一个函数。在 Pyrex 文件中既可以使用“def”来定义函数，也可以使用“cdef”来定义函数。不同的是，使用“def”定义的函数，只能使用 Python 的数据类型，而使用“cdef”定义的函数则可以使用 C 中的数据类型。另外，使用“def”定义的函数可以被模块导出，在 Python 中使用。而使用“cdef”定义的函数则不能被 Python 使用。代码如下所示。

```
cdef int sum(int x, int y):           # 使用 C 数据类型定义函数
    return x + y
cdef sum(x, y):                       # 如果没有声明参数类型，则表示为 Python 中的类型
    return x + y
cdef object sum(object x, object y): # 使用 object 显式地表示参数为 Python 中的类型
    return x + y
```

4. 使用类

使用“cdef”也可以在 Pyrex 文件中定义类。需要注意的是如果通过继承创建类，则父类应该在该 Pyrex 文件中定义，否则应该使用“extern”表示父类在外部定义。代码如下所示。

```
cdef class Stud:                     # 使用 cdef 定义类
    def setname(self, n):            # 定义 setname 方法
        self.name = n
    def showname(self):              # 定义 showname 方法
        print self.name
cdef class Student(Stud):           # 通过继承创建类
    def setnum(self, n):             # 定义 setnum 方法
        self.num = n
    def shownum(self):              # 定义 shownum 方法
        print self.num
```

5. 使用 include 和外部 C 函数

Pyrex 支持类似于 C 语言头文件的“.pxi”文件。使用“.pxi”可以将较大的 Pyrex 文件分成多个文件，这样便于维护。“pxi”文件像 C 语言中的头文件一样，只要使用“include”包含就可以了。除此以外，在 Pyrex 中也可以使用 C 语言头文件中声明的函数原型。例如使用 Python/C API 中的 PyRun_SimpleString 函数，则可以使用如下写法。

```
cdef extern from "Python.h":
    int PyRun_SimpleString(char *command)
```


第 9 章 多线程编程

进程是操作系统中应用程序的执行实例，而线程是进程内部的一个执行单元。当系统创建一个进程后，也就创建了一个主线程。每个进程至少有一个线程，也可以有多个线程。在程序中使用多线程可以实现并行处理，充分利用 CPU。Python 提供了对多线程的支持。在 Python 中可以方便地使用多线程进行编程。

9.1 线程基础

Python 提供了 thread、threading 模块对多线程编程的支持。threading 模块是对 thread 模块的封装。多数情况下应该用 threading 模块来进行多线程编程。

9.1.1 创建线程

在 Python 中可以通过使用 thread 模块中的函数，或者通过继承 threading 类来创建线程。编程创建后还可以对其进行操作。

1. 使用 thread 模块创建线程

thread 模块提供了 start_new_thread 函数，用以创建线程。start_new_thread 函数成功创建线程后将返回线程标识。其函数原型如下所示。

```
start_new_thread( function, args[, kwargs])
```

其参数含义如下。

- function：在线程中执行的函数名。
- args：元组形式的参数列表。
- kwargs：可选参数，以字典的形式指定参数。

以下代码首先定义一个函数，然后使用 start_new_thread 创建一个线程运行该函数。

```
>>> import thread                                # 导入 thread 模块
>>> def run(n):                                   # 定义 run 函数
...     for i in range(n):
...         print i
... 
```

第9章 多线程编程

```

>>> thread.start_new_thread(run, (4,)) # 使用 start_new_thread 函数创建线程
3668 # 返回线程的标识
>>> 0 # 此处为 run 函数的输出
1
2
3
>>> thread.start_new_thread(run, (2,)) # 使用 start_new_thread 函数创建线程
1388 # 返回线程的标识
>>> 0 # 此处为 run 函数的输出
1
>>> thread.start_new_thread(run, (), {'n':4}) # 使用字典向函数传递参数
3896
>>> 0
1
2
3

```

2. 使用 threading 模块创建线程

通过继承 threading 模块中的 Thread 创建新类，重载 run 方法后，可以通过 start 方法创建线程。线程创建后将运行 run 方法。以下代码使用 threading 模块创建线程。

```

>>> import threading # 导入 threading 模块
>>> class mythread(threading.Thread): # 通过继承 Thread 创建类
...     def _init_(self, num): # 定义初始化方法
...         threading.Thread._init_(self) # 调用父类的初始化方法
...         self.num = num
...     def run(self): # 重载 run 方法
...         print 'I am ', self.num
...
>>> t1 = mythread(1) # 生成 mythread 对象
>>> t2 = mythread(2) # 生成 mythread 对象
>>> t3 = mythread(3) # 生成 mythread 对象
>>> t1.start() # 运行线程 t1，实际上是运行 run 方法
I am 1
>>> t2.start() # 运行线程 t2
I am 2
>>> t3.start() # 运行线程 t3
I am 3

```

除了通过继承 threading.Thread 创建类以外，还可以通过使用 threading.Thread 直接在线程中运行函数。以下代码使用 threading.Thread 直接创建线程。

```

>>> import threading # 导入 threading 模块
>>> def run(x, y): # 定义 run 函数
...     for i in range(x, y):
...         print i
...
>>> t1 = threading.Thread(target = run, args = (15, 20)) # 直接使用 Thread 附加函数，args 为函数参数
>>> t1.start() # 运行线程
15

```

```

16
17
18
19
>>> t2 = threading.Thread(target = run, args = (7,11))      # 直接使用 Thread 附加函数,
                                                            # args 为函数参数
>>> t2.start()                                              # 运行线程
7
8
9
10

```

9.1.2 Thread 对象中的方法

在上一节的例子中,仅使用了 Thread 对象中的 start 方法,重载了 Thread 对象的 run 方法。当线程被运行时,将运行 run 方法。Thread 对象还具有以下的几种方法。

1. join 方法

如果一个线程或者在函数的执行过程中调用另一个线程,并且待其完成操作后才能执行,那么在调用线程时可以使用被调用线程的 join 方法。join 方法的原型如下所示。

```
join([timeout])
```

其参数含义如下。

- timeout: 可选参数,线程运行的最长时间。

以下代码使用 Thread 对象的 join 方法等待线程完成操作。

```

>>> import threading      # 导入 threading 模块
>>> import time           # 导入 time 模块
>>> class Mythread(threading.Thread):      # 通过继承 Thread 创建类
...     def _init_(self,id):              # 初始化方法
...         threading.Thread._init_(self) # 调用父类的初始化方法
...         self.id = id
...     def run(self):                    # 重载 run 方法
...         x = 0
...         time.sleep(60)                # 使用 time 模块中的 sleep 方法让线程休眠 60s
...         print self.id
...
>>> def func():                      # 定义函数
...     t.start()                     # 运行线程
...     for i in range(5):
...         print i
...
>>> t = Mythread(2)                  # 生成 Mythread 对象
>>> func()                           # 调用函数,运行线程
0                                     # 输出结果中没有线程的输出,func 函数没有等待
                                     # 线程完成
1
2
3

```


第9章 多线程编程

```

4
>>> def func():
...     t.start()
...     t.join()
...     for i in range(5):
...         print i
...
>>> t = Mythread(3)
>>> func()
3
0
1
2
3
4

```

重新定义 func 函数
执行函数，运行线程
调用 join 方法等待线程完成

生成 Mythread 对象
调用函数，运行线程
此为线程输出

2. isAlive 方法

当线程创建后，可以使用 Thread 对象的 isAlive 方法查看线程是否运行。如下代码使用 Thread 对象的 isAlive 方法查看线程是否运行。

```

>>> import threading
>>> import time
>>> class mythread(threading.Thread):
...     def _init_(self, id):
...         threading.Thread._init_(self)
...         self.id = id
...     def run(self):
...         time.sleep(5)
...         print self.id
...
>>> t = mythread(1)
>>> def func():
...     t.start()
...     print t.isAlive()
...
>>> func()
True
>>> 1

```

导入 threading 模块
导入 time 模块
通过继承 Thread 创建类
初始化方法
调用父类的初始化方法

重载 run 方法

生成 mythread 对象
定义函数
运行线程
打印线程状态

调用函数
线程状态
线程输出

3. 线程名

当线程创建后可以设置线程名来区分不同的线程，以便对线程进行控制。线程名可以在类的初始化函数中定义，也可以使用 Thread 对象 setName 方法设置线程名。使用 Thread 对象的 getName 方法可以获得线程名。以下代码使用不同方法设置线程名。

```

>>> import threading
>>> class mythread(threading.Thread):
...     def _init_(self, threadname):
...         threading.Thread._init_(self, name = threadname)
...     def run(self):
...         print self.getName()

```

导入 threading 模块
通过继承 Thread 创建类

初始化线程名
重载 run 方法

```

...
>>> t1 = mythread('t1')
>>> t1.getName()
't1'
>>> t1.setName('T')
>>> t1.getName()
'T'
>>> t2 = mythread('t2')
>>> t2.start()
t2
>>> t2.getName()
't2'
>>> t2.setName('TT')
>>> t2.getName()
'TT'

```

类实例化, 设置线程名
调用 getName 方法获得线程名
调用 setName 方法设置线程名
调用 getName 方法获得线程名
类实例化, 设置线程名
运行线程
调用 getName 方法获得线程名
调用 setName 方法设置线程名
调用 getName 方法获得线程名

4. setDaemon 方法

在脚本运行过程中有一个主线程, 如果主线程又创建一个子线程, 那么当主线程退出时, 会检验子线程是否完成。如果子线程未完成, 则主线程会等待子线程完成后再退出。当需要主线程退出时, 不管子线程是否完成都随主线程退出, 则可以使用 Thread 对象的 setDaemon 方法来设置。以下代码使用 Thread 对象的 setDaemon 方法设置线程随主线程结束而结束。

```

# -*- coding:utf-8 -*-
# file: threaddaemon.py
#
import threading
import time
class mythread(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        time.sleep(5)

        print self.getName()
def func1():
    t1.start()
    print 'func1 done'
def func2():
    t2.start()
    print 'func2 done'
t1 = mythread('t1')
t2 = mythread('t2')
t2.setDaemon(True)
func1()
func2()

```

导入 threading 模块
导入 time 模块
通过继承创建类
初始化方法
调用父类的初始化方法
重载 run 方法
调用 time.sleep 函数, 让线程休眠 5s
定义函数 func1
定义函数 func2
类实例化
类实例化
设置 t2 的 Daemon 标志
调用函数 func1
调用函数 func2

运行 threaddaemon.py 脚本输出如下所示。

```

func1 done
func2 done
t1

```

由于调用了线程 t2 的 setDaemon 方法, 当主线程结束时, 线程 t2 也随之结束。因此, t2 还没来得及打印自己的线程名, 就已经结束。将 threaddaemon.py 脚本中的 “t2.setDaemon(True)” 删除后, 保存脚本。重新运行 threaddaemon.py 脚本, 脚本输出如下所示。

```
func1 done
func2 done
t1
t2
```

修改后的脚本要等待所有子线程完成后才会退出, 因此, 线程 t1 和线程 t2 都被执行完。如果在交互式模式下运行该实例, 则不会有区别, 因为, 在交互式模式下的主线程只有在退出 Python 时才终止。

9.2 线程同步

如果多个线程共同对某个数据修改, 则可能会出现不可预料的结果。为了保证数据被正确修改, 需要对多个线程进行同步。

9.2.1 简单的线程同步

使用 Thread 对象的 Lock 和 RLock 可以实现简单的线程同步。Lock 对象和 RLock 对象都具有 acquire 方法和 release 方法。对于如果需要每次只有一个线程操作的数据, 可以将操作过程放在 acquire 方法和 release 方法之间。如下 syn.py 脚本使用 acquire 方法和 release 方法保持线程同步。

```
# -*- coding:utf-8 -*-
# file: syn.py
#
import threading
import time
class mythread(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global x
        lock.acquire()
        for i in range(3):
            x = x + 1
            time.sleep(2)
            print x
        lock.release()
lock = threading.RLock()
t1 = []
for i in range(10):
    t = mythread(str(i))

# 导入 threading 模块
# 导入 time 模块
# 通过继承创建类
# 初始化方法
# 调用父类的初始化方法
# 重载 run 方法
# 使用 global 表明 x 为全局变量
# 调用 lock 的 acquire 方法

# 调用 sleep 函数, 让线程休眠 2s

# 调用 lock 的 release 方法
# 生成 RLock 对象
# 定义列表

# 类实例化
```



```

t1.append(t)                                # 将类对象添加到列表中

x=0                                           # 将 x 赋值为 0
for i in t1:
    i.start()                                # 依次运行线程

```

运行脚本后输出如下所示。

```

3
6
9
12
15
18
21
24
27
30

```

将 syn.py 脚本中的第 11 行的“lock.acquire()”、第 16 行的“lock.release()”和第 17 行的“lock = threading.Lock()”删除后保存脚本。运行修改后的脚本输出如下所示。

```

30
30
30
30
30
30
30
30
30
30
30

```

修改后的脚本输出都是 30 也就是 i 的最终值。由于 i 是全局变量，在每个线程对 i 进行操作后就“休眠”了。在线程休眠的时候，Python 解释器已经执行了其他的线程，而使 i 值增加。当所有线程“休眠”结束后，i 的值已被所有线程修改变成了 30，因此输出为 30。

而在使用 Lock 对象的脚本中，对全局变量 i 的操作放在 acquire 方法和 release 方法之间。Python 解释器每次仅允许一个线程对 i 进行操作。只有当该线程操作完后，并且结束休眠以后才开始下一个线程，所以使用 Lock 对象的脚本输出是依次递增的。

9.2.2 使用条件变量保持线程同步

Python 的 Condition 对象提供了对复杂线程同步的支持。使用 Condition 对象可以在某些事件触发后才处理数据。Condition 对象除了具有 acquire 方法和 release 方法以外，还有 wait 方法、notify 方法、notifyAll 方法等用于条件处理。以下 P_C.py 脚本使用 Condition 对象来实现著名的生产者和消费者的关系。

```

# -*- coding:utf-8 -*-
# file: P_C.py

```

第9章 多线程编程

```

#
import threading                                     # 导入 threading 模块
class Producer(threading.Thread):                     # 定义生产者类
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global x
        con.acquire()                                # 调用 con 的 acquire 方法
        if x == 1000000:
            con.wait()                                # 调用 con 的 wait 方法
            pass
        else:
            for i in range(1000000):
                x = x + 1
                con.notify()                           # 调用 con 的 notify 方法
            print x
            con.release()                             # 调用 con 的 release 方法
class Consumer(threading.Thread):                     # 定义生产者类
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global x
        con.acquire()
        if x == 0:
            con.wait()
            pass
        else:
            for i in range(1000000):
                x = x - 1
                con.notify()
            print x
            con.release()
con = threading.Condition()                           # 生成 Condition 对象
x=0
p = Producer('Producer')                             # 生成生产者对象
c = Consumer('Consumer')                             # 生成消费者对象
p.start()                                              # 运行线程
c.start()
p.join()                                              # 等待线程结束
c.join()
print x

```

运行脚本后输出如下所示。

```

1000000
0
0

```

- 将脚本中第 11、13、18、26、28、33、36 等行中使用 Condition() 对象的语句删除。运行修改后的脚本输出如下所示。

```

964166

```

```
-482930
-482930
```

修改后的脚本每次运行后输出可能不一样。

9.2.3 使用队列保持线程同步

Python 中的 Queue 对象也提供了对线程同步的支持。使用 Queue 对象可以实现多生产者和多消费者形成的先进先出的队列。每个生产者将数据依次存入队列，而每个消费者则依次从队列中取出数据。以下 MP_MC.py 脚本使用 Queue 对象实现多个生产者和消费者的关系。

```
# -*- coding:utf-8 -*-
# file: MP_MC.py
#
import threading                                     # 导入 threading 模块
import time                                           # 导入 time 模块
import Queue                                          # 导入 Queue 模块
class Producer(threading.Thread):                    # 定义生产者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global queue                                  # 声明为全局 Queue 对象
        queue.put(self.getName())                    # 调用 put 方法将线程名添加到队列中

        print self.getName(),'put ',self.getName(),' to queue'
class Consumer(threading.Thread):                    # 定义消费者类
    def __init__(self,threadname):
        threading.Thread.__init__(self,name = threadname)
    def run(self):
        global queue
        print self.getName(),'get ',queue.get(),'from queue' # 调用 get 方法获取队列中内容
queue = Queue.Queue()                                # 生成队列对象
plist = []                                           # 生产者对象列表
clist = []                                           # 消费者对象列表
for i in range(10):
    p = Producer('Producer' + str(i))
    plist.append(p)                                  # 添加到生产者对象列表
for i in range(10):
    c = Consumer('Consumer' + str(i))
    clist.append(c)                                  # 添加到消费者对象列表
for i in plist:
    i.start()                                         # 运行生产者线程
    i.join()
for i in clist:
    i.start()                                         # 运行消费者线程
    i.join()
```

运行脚本后输出如下所示。

```
Producer0 put Producer0 to queue
Producer1 put Producer1 to queue
```



```
Producer2 put Producer2 to queue
Producer3 put Producer3 to queue
Producer4 put Producer4 to queue
Producer5 put Producer5 to queue
Producer6 put Producer6 to queue
Producer7 put Producer7 to queue
Producer8 put Producer8 to queue
Producer9 put Producer9 to queue
Consumer0 get Producer0 from queue
Consumer1 get Producer1 from queue
Consumer2 get Producer2 from queue
Consumer3 get Producer3 from queue
Consumer4 get Producer4 from queue
Consumer5 get Producer5 from queue
Consumer6 get Producer6 from queue
Consumer7 get Producer7 from queue
Consumer8 get Producer8 from queue
Consumer9 get Producer9 from queue
```

9.3 线程间通信

Python 提供了 Event 对象用于线程间的相互通信。实际上线程同步在一定程度上已经实现线程间的通信。线程同步是每次仅有一个线程对共享数据进行操作，其他线程则等待。而 Event 对象是由线程设置的信号标志，如果信号标志为真，则其他线程等待直到信号解除。

9.3.1 Event 对象的方法

Event 对象实现了简单的线程通信机制，它提供了设置信号、清除信号、等待等用于实现线程间的通信。

1. 设置信号

使用 Event 对象的 set()方法可以设置 Event 对象内部的信号标志为真。Event 对象提供了 isSet()方法来判断其内部信号标志的状态。当使用 Event 对象的 set()方法后，isSet()方法返回真。

2. 清除信号

使用 Event 对象的 clear()方法可以清除 Event 对象内部的信号标志，即将其设置为假。当使用 Event 对象的 clear()方法后，isSet()方法返回假。

3. 等待

Event 对象 wait 的 wait 方法只有在其内部信号为真时才会很快地执行完成并返回。当 Event 对象的内部信号标志为假时，则 wait 方法一直等到其为真时才返回。另外还可以向 wait 方法传递参数，以设定最长等待时间。

9.3.2 使用 Event 对象实现线程间通信

配合使用 Event 对象的几种方法可以实现进程间的简单通信。以下的 event.py 脚本即使使用 Event 对象实现进程间的通信。

```
# -*- coding:utf-8 -*-
# file: event.py
#
import threading                                # 导入 threading 模块
class mythread(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global event
        if event.isSet():                        # 使用全局 Event 对象
            event.clear()                       # 判断 Event 对象内部信号标志
            event.wait()                        # 若已设置标志则清除
            print self.getName()               # 调用 wait 方法
        else:
            print self.getName()
            event.set()                        # 设置 Event 对象内部信号标志
event = threading.Event()                      # 生成 Event 对象
event.set()                                   # 设置 Event 对象内部信号标志
tl = []
for i in range(10):
    t = mythread(str(i))
    tl.append(t)                                # 生成线程列表

for i in tl:
    i.start()                                  # 运行线程
```

运行脚本后输出如下所示。

```
1
0
3
2
5
4
7
6
9
8
```

9.4 微线程——Stackless Python

Stackless Python 是 Python 的一个增强版本。Stackless Python 修改了 Python 的代码，提供了对微线程的支持。微线程是轻量级的线程，与前边所讲的线程相比，微线程在多个线程

间切换所需的时间更多，占用资源也更少。

9.4.1 Stackless Python 概述

Stackless Python 不是必需的，它只是 Python 的一个修改版本，对多线程编程有更好的支持。如果在对线程应用有较高的要求时可以考虑使用 Stackless Python 来完成。

1. Stackless Python 安装

在安装 Stackless Python 之前应该先安装 Python，根据所安装的 Python 版本到 Stackless Python 的官方网站 <http://www.stackless.com> 下载相应的版本。对于 Windows 有预编译好的 Stackless Python。以 Python 2.5 为例，下载相应的 Stackless Python 版本的压缩包，安装步骤如下所示。

(1) 将压缩包中的 python25.dll 及 python25_d.dll 复制到 Windows 安装目录下的 system32 目录中，替换原有的 python25.dll 及 python25_d.dll。注意在替换前应将原始的文件做好备份，以便在出现错误时恢复。

(2) 将压缩包中 libs 目录中的文件复制到 Python 安装目录下的 libs 目录中，替换原有的文件。

(3) 将压缩包中 Lib 目录中的文件复制到 Python 安装目录下的 Lib 目录中，替换原有的文件。

安装完成后可以在 Python 的交互式环境中输入如下所示代码。

```
import stackless
```

如果没有错误产生，则表示 Stackless Python 已经安装好了。若出现错误，则可能是 Stackless Python 与当前的 Python 版本不兼容，可以考虑使用其他版本的 Python。

2. stackless 模块中的 tasklet 对象

Stackless Python 提供了 stackless 内置模块。stackless 模块中的 tasklet 对象完成了与创建线程类似的功能。使用 tasklet 对象可以像创建线程运行函数那样来运行函数。以下实例使用 tasklet 对象的部分方法运行函数。

```
>>> import stackless                                # 导入 stackless 模块
>>> def show():                                     # 定义 show 函数
...     print 'Stackless Python'
...
>>> st = stackless.tasklet(show)()                  # 调用 tasklet 添加函数，第 2 个括号为函数参数
>>> st.run()                                         # 调用 run 方法，执行函数
Stackless Python
>>> st = stackless.tasklet(show)()                  # 重新生成 st
>>> st.alive                                         # 查看其状态
True
>>> st.kill()                                       # 调用 kill 方法结束线程
>>> st.alive                                         # 查看其状态
False
```



```
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011DD3F0>
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()
Stackless Python
Stackless Python
```

直接调用 tasklet

调用模块的 run 方法

3. stackless 模块中的 schedule 对象

stackless 模块中的 schedule 对象可以控制任务的执行顺序。当有多个任务时，可以使用 schedule 对象使其依次执行。如下代码使用 schedule 对象控制任务顺序。

```
>>> import stackless
>>> def show():
...     stackless.schedule()
...     print 1
...     stackless.schedule()
...     print 2
...
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011CF830>
>>> stackless.tasklet(show)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()
1
1
2
2
```

导入 stackless 模块

定义 show 函数

使用 schedule 控制任务顺序

调用 tasklet，生成任务列表

执行任务

4. stackless 模块中的 channel 对象

使用 stackless 模块中的 channel 对象可以在不同的人之间进行通信，这和线程间的通信类似。使用 channel 对象的 send 方法可以发送数据。使用 channel 对象的 receive 方法可以接收数据。

```
>>> import stackless
>>> def send():
...     chn.send('Stackless Python')
...     print 'I send: Stackless Python'
...
>>> def rec():
...     print 'I receive:', chn.receive()
...
>>> stackless.tasklet(send)()
<stackless.tasklet object at 0x011DD6B0>
>>> stackless.tasklet(rec)()
<stackless.tasklet object at 0x011DD570>
>>> stackless.run()
I receive: Stackless Python
I send: Stackless Python
```

导入 stackless 模块

定义 send 方法

调用 channel 对象的 send 方法发送数据

定义 rec 方法

调用 channel 对象的 receive 方法接收数据

调用 tasklet，生成任务列表

执行任务

9.4.2 使用微线程

使用 Stackless Python 的内置模块 stackless 也可以完成多线程编程,使用起来更加方便。以下 S_P_C.py 脚本将前边生产者与消费者的代码改写为 Stackless 版,代码更加简洁。

```
# -*- coding:utf-8 -*-
# file: S_P_C.py
#
import stackless
import Queue

def Producer(i):
    global queue
    queue.put(i)
    print 'Producer',i, 'add',i
def Consumer():
    global queue
    i = queue.get()
    print 'Consumer',i, 'get',i
queue = Queue.Queue()
for i in range(10):
    stackless.tasklet(Producer)(i)
for i in range(10):
    stackless.tasklet(Consumer)()
stackless.run()
```

导入 stackless 模块
导入 Queue 模块
定义生产者
声明为全局 Queue 对象
向队列中添加数据

定义消费者

从队列中取出数据

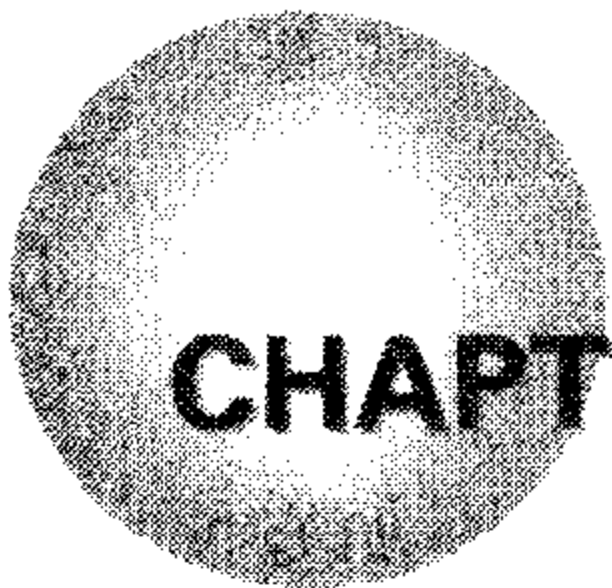
生成队列对象

添加生产者任务

添加消费者任务
执行任务

运行脚本后输出如下所示。

```
Producer 0 add 0
Producer 1 add 1
Producer 2 add 2
Producer 3 add 3
Producer 4 add 4
Producer 5 add 5
Producer 6 add 6
Producer 7 add 7
Producer 8 add 8
Producer 9 add 9
Consumer 0 get 0
Consumer 1 get 1
Consumer 2 get 2
Consumer 3 get 3
Consumer 4 get 4
Consumer 5 get 5
Consumer 6 get 6
Consumer 7 get 7
Consumer 8 get 8
Consumer 9 get 9
```



CHAPTER 10

第 10 章 系统编程

Python 虽然是脚本语言，但借助其扩展一样可以进行系统级别的程序编写。在这一章中，将使用 PythonWin 提供的 Windows API 函数接口编写与系统相关的 Python 脚本。与使用 VC++ 编写的应用程序在功能上并没有区别，而且使用 Python 省去了编译、链接的过程。使用 Python 开发一些实用的脚本更为迅速，在代码上更加简洁。

10.1 访问 Windows 注册表

通过使用 win32api 模块，Python 可以方便地访问注册表，并对其进行打开、关闭、添加项、删除项，以及添加、修改项值等操作。

10.1.1 注册表概述

Windows 注册表是 Windows 系统用于存储用户、应用程序和硬件设备配置系统所必需信息的数据库。Windows 注册表中包含了系统在运行期间需要查询的信息，如用户的配置文件、系统中的应用程序以及计算机的硬件信息。Windows 注册表中的信息影响着系统的运行，因此注册表也成了 Windows 系统中的“是非之地”。Windows 注册表由 5 个基本项组成，如表 10-1 所示。

表 10-1

Windows 注册表基本项

项 名	描 述
HKEY_CLASSES_ROOT	是 HKEY_LOCAL_MACHINE\Software 的子项，保存打开文件所对应的应用程序信息
HKEY_CURRENT_USER	是 HKEY_USERS 的子项，保存当前用户的配置信息
HKEY_LOCAL_MACHINE	保存计算机的配置信息，针对所有用户
HKEY_USERS	保存计算机上的所有以活动方式加载的用户配置文件
HKEY_CURRENT_CONFIG	保存计算机的硬件配置文件信息

对 Windows 注册表进行操作的基本步骤是：首先导入 win32api 和 win32con 模块，打开

要进行操作的项，获得该项的句柄，然后执行相关的操作，如读、写、修改等，完成操作后，需要关闭注册表，以释放资源，如图 10-1 所示。由于注册表中存放着系统的重要数据，对注册表进行操作前应备份注册表，以避免误操作导致系统崩溃。一个简单的例子如下所示。

```
>>> import win32api    #导入模块
>>> import win32con
# 项的具体值由安装的 Python 版本决定
>>> name = 'SOFTWARE\\Python\\PythonCore\\2.5\\InstallPath'
# 打开注册表，获得要进行操作的项的句柄
>>> key = win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE,name,0,win32con.KEY_ALL_ACCESS)
>>> win32api.RegQueryValue(key, '')    # 进行操作，这里读取项的默认值
'D:\\Python25\\'                      # 输出项的默认值
>>> win32api.RegCloseKey(key)          # 关闭注册表，结束操作
```

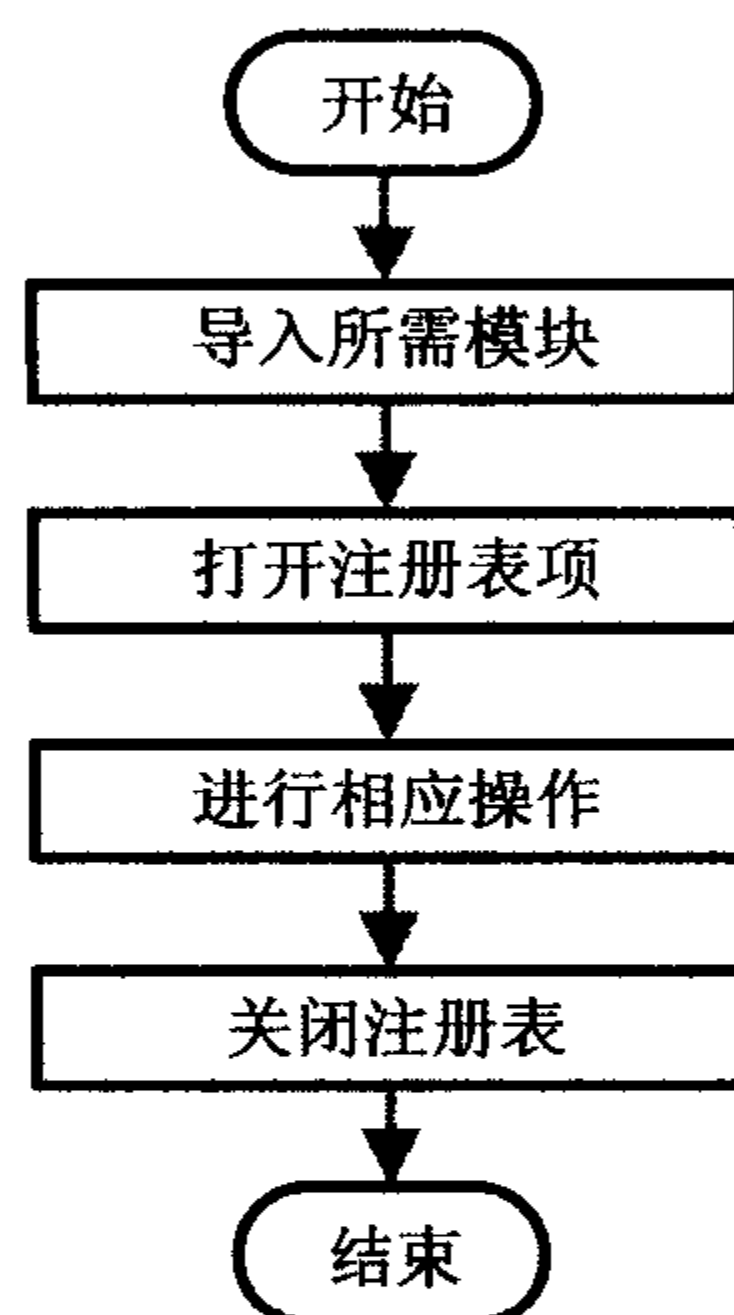


图 10-1 注册表操作基本流程

10.1.2 使用 Python 操作注册表

与注册表操作相关的函数可以分为打开注册表、关闭注册表、读取项值、设置添加项值、添加项，以及删除项等几类。

1. 打开注册表

对注册表进行操作前，必须打开注册表。在 Python 中，可以使用以下两个函数：RegOpenKey 和 RegOpenKeyEx。其函数原型分别如下所示。

```
RegOpenKey(key, subKey, reserved, sam)
RegOpenKeyEx(key, subKey, reserved, sam)
```

两个函数的参数一样。参数含义如下。

- key：必须为表 10-1 中列出的项。
- subKey：要打开的子项。
- reserved：必须为 0。

- sam：对打开的子项进行的操作，包括 win32con.KEY_ALL_ACCESS、win32con.KEY_READ、win32con.KEY_WRITE 等。

以下实例实现打开注册表“HKEY_CURRENT_USER\Software”项。

```
>>> import win32api      # 导入 win32api 模块
>>> import win32con      # 导入 win32con 模块
# 使用 RegOpenKey 打开注册表项
>>> key = win32api.RegOpenKey(win32con.HKEY_CURRENT_USER, 'Software', 0, win32con.KEY_READ)
>>> print key             # key 为打开的项的句柄
<PyHKEY:260>
```

2. 关闭注册表

打开的注册表，在操作完成后，需要关闭。在 Python 中使用 RegCloseKey 函数关闭打开的注册表项。其函数原型如下所示。

```
RegCloseKey(key)
```

其参数只有一个，其含义如下。

- key：已经打开的注册表项。

以下实例关闭一个已经打开的注册表项。

```
# 关闭刚才打开的注册表项
>>> win32api.RegCloseKey(key)
>>> print key
<PyHKEY:0>
```

3. 读取项值

在打开注册表项以后，可以使用 RegQueryValue 函数读取项的默认值。如果要读取某一项值，可以使用 RegQueryValueEx 函数。其函数原型分别如下所示。

```
RegQueryValue(key, subKey )
RegQueryValueEx(key, valueName )
```

对于 RegQueryValue，其参数含义如下。

- key：已打开的注册表项的句柄。
- subKey：要操作的子项。

对于 RegQueryValueEx，其参数含义如下。

- key：已经打开的注册表项的句柄。
- valueName：要读取的项值名称。

以下实例实现对“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”项的操作。

```
>>> import win32api
>>> import win32con
# 打开“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”项
>>> key = win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE, 'SOFTWARE\\Microsoft\\Internet Explorer', 0, win32con.KEY_ALL_ACCESS)
>>> win32api.RegQueryValue(key, '')    # 读取项的默认值
```

```

''                                     # 输出为空, 表示其默认值未设置
# 读取项值名称为 Version 的项值数据, 也就是 Internet Explorer 的版本
>>> win32api.RegQueryValueEx(key, 'Version')
('6.0.2900.2180', 1)
>>> win32api.RegQueryInfoKey(key)      # RegQueryInfoKey 函数查询项的基本信息
(26, 7, 128178812229687500L)          # 返回项的子项数目、项值数目, 以及最后一次修改时间

```

4. 设置项值

要修改或者重新设置注册表某一项的项值可以使用 RegSetValueEx 函数, 如果要设置项的默认值可以使用 RegSetValue 函数。需要说明的是, 对于 RegSetValueEx, 如果要设置的项值不存在, 那么 RegSetValueEx 会添加该项值, 如果存在, 则修改该项值。其函数原型分别如下所示。

```

RegSetValueEx(key, valueName, reserved, type, value)
RegSetValue(key, subKey, type, value)

```

对于 RegSetValueEx, 其参数含义如下。

- key: 要设置的项的句柄。
- valueName: 要设置的项值名称。
- reserved: 保留, 可以设为 0。
- type: 项值的类型。
- value: 所要设置的值。

对于 RegSetValue, 其参数的含义如下。

- key: 已经打开的项的句柄。
- subKey: 所要设置的子项。
- type: 项值的类型, 必须为 win32con.REG_SZ。
- value: 项值数据, 为字符串。

以下的实例实现修改 “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer” 的默认值, 以及其 “Version” 项值数据。

```

# 将 “HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer” 的默认值设为 python
>>> win32api.RegSetValue(key, '', win32con.REG_SZ, 'python')
# 将其 “Version” 设置为 7.0.2900.2180
>>> win32api.RegSetValueEx(key, 'Version', 0, win32con.REG_SZ, '7.0.2900.2180')

```

5. 添加、删除项

要向注册表中添加项可以使用 RegCreateKey 函数。RegDeleteKey 函数可以删除注册表中的项。其参数原型分别如下所示。

```

RegCreateKey(key, subKey)
RegDeleteKey(key, subKey)

```

其参数含义相同, 参数含义分别如下。

- key: 已经打开的注册表项的句柄。

- subKey: 所要操作（添加或删除）的子项。

以下的实例实现对“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”项的添加、删除子项操作。

```
# 向“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer”添加子项“Python”
>>> win32api.RegCreateKey(key, 'Python')
<PyHKEY:336> # 新创建的子项的句柄
# 删除刚才创建的子项“Python”
>>> win32api.RegDeleteKey(key, 'Python')
```

10.1.3 查看系统启动项

很多流氓软件、木马，以及病毒等都会随着系统的启动而运行，它们一般都采取修改注册表的方法，将自身路径添加到注册表中，以达到开机运行的目的。

下边的 Python 脚本通过查询注册表中以下几项的值，可以列举出随系统启动的可执行文件，便于查找可疑项目，从而清除系统中的恶意软件。

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run。
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce。
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx。
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run。
- HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce。

```
# -*- coding: iso-8859-1 -*-
# file: AutoRuns.py
#
import string # 导入所需要的模块
from win32api import *
from win32con import *
def GetValues(fullname): # GetValues 函数用于获得某注册表项下所有的项值
    name = string.split(fullname, '\\', 1) # 把完整的项拆分成根项和子项两部分
    # 打开相应的项，为了让该函数更通用
    # 使用了多个判断语句
    if name[0] == 'HKEY_LOCAL_MACHINE':
        key = RegOpenKey(HKEY_LOCAL_MACHINE, name[1], 0, KEY_READ)
    elif name[0] == 'HKEY_CURRENT_USER':
        key = RegOpenKey(HKEY_CURRENT_USER, name[1], 0, KEY_READ)
    elif name[0] == 'HKEY_CLASSES_ROOT':
        key = RegOpenKey(HKEY_CLASSES_ROOT, name[1], 0, KEY_READ)
    elif name[0] == 'HKEY_CURRENT_CONFIG':
        key = RegOpenKey(HKEY_CURRENT_CONFIG, name[1], 0, KEY_READ)
    elif name[0] == 'HKEY_USERS':
        key = RegOpenKey(HKEY_USERS, name[1], 0, KEY_READ)
    else:
        print 'err, no key named %s' (name[0])
    info = RegQueryInfoKey(key) # 查询项的项值数目
```

```

# 遍历项值获得项值数据
for i in range(0,info[1]):
    ValueName = RegEnumValue(key, i)
    print string.ljust(ValueName[0],20),ValueName[1]    # 调整项值名称长度,使输出更好看
    RegCloseKey(key)                                    # 关闭打开的项
# 因为 GetValues 函数比较通用,所以可以在其他脚本中调用
# 这里先检查脚本是否被其他脚本调用
if __name__ == '__main__':
    # 因为要检查的项较多,故将其放在列表中,便于增减
    KeyNames = ['HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run',\
                'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce',\
                'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnceEx',\
                'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run',\
                'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce']
    for KeyName in KeyNames:                            # 遍历列表,调用 GetValues 函数,
                                                         输出项值
        print KeyName
        GetValues(KeyName)

```

10.1.4 修改 IE

IE 浏览器的相关设置也保存在注册表里,只要通过设置注册表中相应的项值,就可以修改 IE 的主页、标题栏等。与 IE 设置相关的几个注册表项如下所示。

- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Start Page 项保存的是 IE 的主页地址。
- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Window Title 项保存的是 IE 的标题栏。
- HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\Search Page 项保存的是 IE 默认搜索页。

为了打造一个个性化的 IE,可以编写一个 Python 脚本。利用上节的知识将其设置为开机自动运行,然后通过脚本获得当前的日期,并随机选择预先设置好的个性标题,这样就可以打造每天都不同的个性化 IE。还可以使用 Python 的网络编程功能获取天气预报网站上的天气情况,也将其添加到 IE 的标题栏。这里暂时先不实现这个功能,在完成后面章节的学习后,再将获取天气情况的功能添加到该脚本中。

首先编写将脚本设置为开机运行的 Add2AutoRun.py,其代码如下:

```

# -*- coding: iso-8859-1 -*-
# file: Add2AutoRun.py
#
import win32api
import win32con

```

```

name = 'SetIE' # 要添加的项值名称
path = 'C:\\SetIE.py' # 要添加的 Python 脚本的路径

KeyName = 'Software\\Microsoft\\Windows\\CurrentVersion\\Run' # 注册表项名
# 异常处理
try:
    # 打开注册表项
    key = win32api.RegOpenKey(win32con.HKEY_CURRENT_USER, \
                               KeyName, \
                               0, \
                               win32con.KEY_ALL_ACCESS)
    win32api.RegSetValueEx(key, name, 0, win32con.REG_SZ, path) # 设置项值
    win32api.RegCloseKey(key) # 关闭注册表
except:
    print 'error'
print 'added that!'

```

如果要添加其他的脚本，只需要修改变量 name 以及 path，分别设置为启动项目的名字和脚本所在的路径即可。以下代码实现修改 IE 的主页、打造个性化 IE 标题栏等功能。

```

# -*- coding: iso-8859-1 -*-
# file: SetIE.py
#
import datetime
import string
import win32api
import win32con
keyname = 'Software\\Microsoft\\Internet Explorer\\Main' # 要修改的注册表项
page = 'www.python.org' # 要设置为主页的网址
today = datetime.date.today() # 获取当前日期
# 将日期格式化为 xxxx 年 xx 月 xx 日的形式
title = today.strftime('%Y')+ '年'+today.strftime('%m')+ '月'+today.strftime('%d')+ '日'
# 异常处理
try:
    # 打开注册表项，获得句柄
    key = win32api.RegOpenKey(win32con.HKEY_CURRENT_USER, keyname, 0, win32con.
KEY_ALL_ACCESS)
    # 读取"Start Page"的项值数据
    StartPage = win32api.RegQueryValueEx(key, 'Start Page')
except:
    print 'error'
else:
    # 判断主页是否为要修改的主页，如果不是则修改
    if StartPage[0] != page:
        win32api.RegSetValueEx(key, 'Start Page', 0, win32con.REG_SZ, page)
    # 设置 IE 的标题栏为 xxxx 年 xx 月 xx 日
    win32api.RegSetValueEx(key, 'Window Title', 0, win32con.REG_SZ, title)
    win32api.RegCloseKey(key) # 关闭注册表

```

脚本运行后，可以看到 IE 浏览器的标题栏被设置为 xxxx 年 xx 月 xx 日的形式，如图 10-2 所示。



图 10-2 修改后的 IE 界面

10.2 文件和目录

在系统中不免与文件和目录打交道。对于一些比较繁琐的文件和目录操作，可以使用 Python 提供的 `os` 模块。`os` 模块中包含很多操作文件和目录的函数，可以方便地重命名文件，添加、删除、复制目录以及文件等。

10.2.1 文件目录常用函数

在进行文件和目录操作的时候，一般会用到以下几种操作。

1. 获得当前路径

在 Python 中可以使用 `os.getcwd()` 函数获得当前的路径。其原型如下所示。

```
os.getcwd()
```

该函数不需要传递参数，它返回当前的目录。需要说明的是，当前目录并不是指脚本所在的目录，而是所运行脚本的目录。例如，在 PythonWin 中输入如下脚本。

```
>>> import os
>>> print 'current directory is ',os.getcwd()
current directory is D:\Python25\Lib\site-packages\pythonwin #这里是 PythonWin 的安装目录
```

如果将上述内容写入 `pwd.py`，假设 `pwd.py` 位于 `E:\book\code` 目录，运行 Windows 的命令窗口，进入 `E:\book` 目录，输入 `code\pwd.py`，输出如下所示。

```
E:\book>code\pwd.py
current directory is E:\book
```

2. 获得目录中的内容

在 Python 中可以使用 `os.listdir()` 函数获得指定目录中的内容。其原型如下所示。

`os.listdir(path)`

其参数含义如下。

- `path` 要获得内容目录的路径。

以下实例获得当前目录的内容。

```
>>> import os
>>> os.listdir(os.getcwd())                            # 获得当前目录中的内容
['dde.pyd', 'license.txt', 'Pythonwin.exe', 'scintilla.dll', 'win32ui.pyd',
'win32uiole.pyd', 'pywin']
```

3. 创建目录

在 Python 中可以使用 `os.mkdir()` 函数创建目录。其原型如下所示。

`os.mkdir(path)`

其参数含义为。

- `path` 要创建目录的路径。

以下的实例将在 `E:\book` 目录下创建 `temp` 目录。

```
>>> import os
>>> os.mkdir('E:\\book\\temp')                        # 使用 os.mkdir 创建目录
```

4. 删除目录

在 Python 中可以使用 `os.rmdir()` 函数删除目录。其原型如下所示。

`os.rmdir(path)`

其参数含义如下。

- `path` 要删除的目录的路径。

以下实例删除 `E:\book\temp` 目录。

```
>>> import os
>>> os.rmdir('E:\\book\\temp')                        # 删除目录
```

需要说明的是，使用 `os.rmdir` 删除的目录必须为空目录，否则函数出错。

5. 判断是否是目录

在 Python 中可以使用 `os.path.isdir()` 函数判断某一路径是否为目录。其函数原型如下所示。

`os.path.isdir(path)`

其参数含义如下。

- `path` 要进行判断的路径。

以下实例判断 `E:\book\temp` 是否为目录。

```
>>> import os
>>> os.path.isdir('E:\\book\\temp')                    # 判断 E:\book\temp 是否为目录
True                                                    # 表 E:\book\temp 是目录
```

6. 判断是否为文件

在 Python 中可以使用 `os.path.isfile()` 函数判断某一路径是否为文件。其函数原型如下所示。

```
os.path.isfile(path)
```

其参数含义如下。

- path: 要进行判断的路径。

以下实例判断 E:\book\temp 是否为文件。

```
>>> import os
>>> os.path.isfile('E:\\book\\temp') # 判断是否为文件
False # 表示 E:\book\temp 不是文件
```

10.2.2 批量重命名

在日常工作中经常会遇到这样的情况，需要将某个文件夹下的文件按照一定的规律重新命名。如果手工完成的话，需要耗费大量的时间，而且容易出错。在学习 Python 以后，完全可以写一个简单的脚本完成这样的工作。

```
import os
prefix = 'Python' # prefix 为重命名后的文件起始字符
length = 2 # length 为除去 prefix 后，文件名要达到的长度
base = 1 # 文件名的起始数
format = 'mdb' # 文件的后缀名
# 函数 PadLeft 将文件名补全到指定长度
# str 为要补全的字符
# num 为要达到的长度
# padstr 为达到长度所添加的字符
def PadLeft(str, num, padstr):
    stringlength = len(str)
    n = num - stringlength
    if n >= 0 :
        str = padstr * n + str
    return str
# 为了避免误操作，这里先提示用户
print 'the files in %s will be renamed' % os.getcwd()
input = raw_input('press y to continue\n') # 获取用户输入
if input != 'y': # 判断用户输入，以决定是否执行重命名操作
    exit()
filenames = os.listdir(os.getcwd()) # 获得当前目录中的内容
# 从基数减 1，为了使下边 i = i + 1 在第一次执行时等于基数
i = base - 1
for filename in filenames: # 遍历目录中内容，进行重命名操作
    i = i + 1
    # 判断当前路径是否为文件，并且不是“rename.py”
    if filename != "rename.py" and os.path.isfile(filename):
        name = str(i) # 将 i 转换成字符
        name = PadLeft(name, length, '0') # 将 name 补全到指定长度
        t = filename.split('.') # 分割文件名，以检查其是否是所要修改的类型
        m = len(t)
        if format == '': # 如果未指定文件类型，则更改当前目录中所有文件
            os.rename(filename, prefix + name + '.' + t[m-1])
        else: # 否则只修改指定类型
```



```

        if t[m-1] == format:
            os.rename(filename, prefix+name+'.'+t[m-1])
        else:
            i = i - 1                # 保证 i 连续
    else:
        i = i - 1                # 保证 i 连续

```

10.2.3 代码框架生成器

编写代码要养成良好的习惯，为了使脚本更具可读性，往往需要添加注释，而且还应该在脚本头添加基本的说明，如作者、文件名、日期、用途、版权说明，以及所需要使用的模块等信息。这样，不仅便于保存脚本，而且也便于交流。

但是，如果每次编写一个脚本就依次添加这样的信息，不免有些麻烦，以下代码实现了一个简单的代码框架生成器。

```

# -*- coding:utf-8 -*-
# file: MakeCode.py
#
import os
import sys
import string
import datetime
# python 脚本模板
py = '''#-----
# TO:
#-----
# BY:
#-----
'''
# c 模板
c = '''#-----
* TO:
#-----
* BY:
#-----
'''

if os.path.isfile(sys.argv[1]):                # 判断要创建的文件是否存在，如果存在则退出脚本
    print '%s already exist!' % sys.argv[1]
    sys.exit()
file = open(sys.argv[1], 'w')                    # 创建文件
today = datetime.date.today()                    # 获得当前日期，并格式化为 xxxx-xx-xx 的形式
date = today.strftime('%Y')+ '-' + today.strftime('%m')+ '-' + today.strftime('%d')
filetypes = string.split(sys.argv[1], '.') # 判断将创建的文件是什么类型以便对其分别处理
length = len(filetypes)
filetype = filetypes[length - 1]
if filetype == 'py':
    print 'use python mode'
    file.writelines('# -*- coding:utf-8 -*-')

```

```

file.write('\n')
file.writelines('# File: ' + sys.argv[1])
file.write('\n')
file.write(py)
file.write('# Date: ' + date)
file.write('\n')
file.write('#-----')
elif filetype == 'c' or filetype == 'cpp':
    print 'use c mode'
    file.writelines('/*')
    file.write('\n')
    file.writelines(' *-----')
    file.write('\n')
    file.writelines(' * File: ' + sys.argv[1])
    file.write('\n')
    file.write(c)
    file.write(' * Date: ' + date)
    file.write('\n')
    file.write(' *-----')
    file.write('\n')
    file.write(' */ \n')
else:
    print 'just create %s' % sys.argv[1]
file.close() # 关闭文件

```

写好的脚本可以放到 Windows 的系统目录中，这样就可以随时运行，方便地在目录中产生 Python 脚本或者 C/C++ 文件。

10.3 使用 py2exe 生成可执行文件

使用 Python 固然方便，但不足之处是必须在本机安装 Python 解释器。如果本机没有安装 Python，而又需要运行脚本，可以使用 py2exe 在安装了 Python 的机器上，将 Python 脚本编译成 Windows 可执行文件，这样就不需要 Python 解释器了。

10.3.1 安装 py2exe

py2exe 可以从其官方网站 <http://www.py2exe.org> 下载。下载时应注意根据所安装的版本选择相应的安装程序。py2exe 的安装十分简单，其安装程序会自动搜索 Python 的安装目录。双击运行安装程序，单击下一步，如果安装程序搜索到 Python 的安装路径，会出现如图 10-3 所示的界面。只要一直单击下一步，即可完成安装。安装完成后可以在 Python Shell 中输入如下所示代码。

```

>>> import py2exe
# 如果有如下输出表示未安装成功，如果没有输出的话，则表示 py2exe 模块已经安装好
Traceback (most recent call last):

```

```
File "<interactive input>", line 1, in <module>
ImportError: No module named py2exe
```

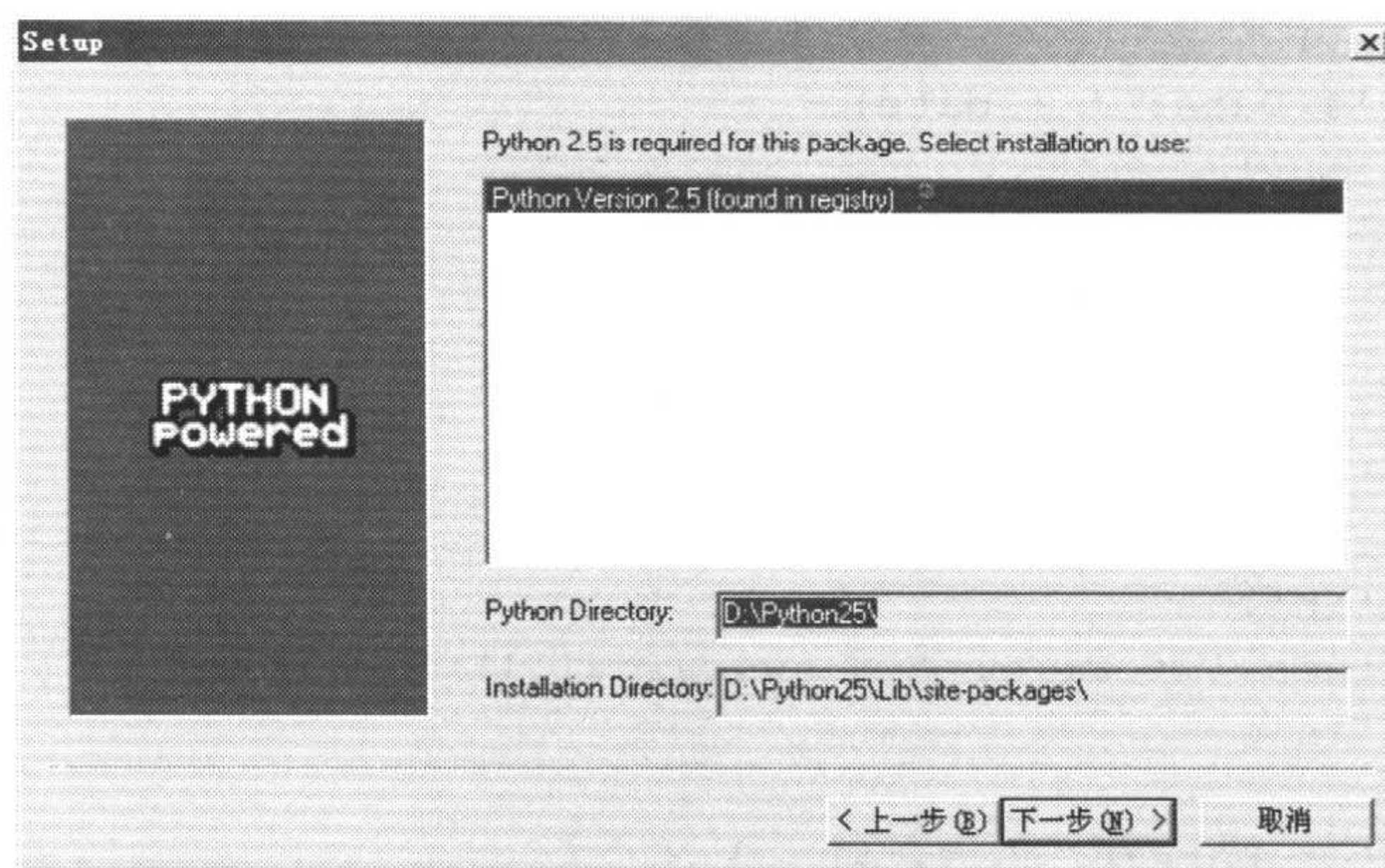


图 10-3 安装 py2exe

10.3.2 使用 py2exe 生成可执行文件

要使用 py2exe，首先要编写一个编译脚本，然后通过 Python 运行编译脚本即可将其他的脚本编译成可执行文件。以下实例是将要编译成可执行文件的脚本。

```
#file: MessageBox.py
import win32api
import win32con
win32api.MessageBox(0, 'hi!!', 'Python', win32con.MB_OK)
```

以下实例是编译脚本。

```
#file: setup.py
import distutils
import py2exe
distutils.core.setup(windows=['MessageBox.py'])
```

在 Windows 的 cmd 窗口中输入“setup.py py2exe”，将得到如下输出：

```
running py2exe
*** searching for required modules ***
*** parsing results ***
creating python loader for extension 'win32api'
creating python loader for extension 'unicodedata'
creating python loader for extension 'bz2'
*** finding dlls needed ***
*** create binaries ***
*** byte compile python files ***
*****
                        部分输出省略
*****
byte-compiling E:\book\code\py2exe\build\bdist.win32\winexe\temp\unicodedata.py
to unicodedata.pyc
```



```

byte-compiling E:\book\code\py2exe\build\bdist.win32\winexe\temp\win32api.py to
win32api.pyc
*** copy extensions ***
*** copy dlls ***
copying D:\Python25\lib\site-packages\py2exe\run_w.exe -> E:\book\code\py2exe\dist\MessageBox.exe

```

```

*** binary dependencies ***

```

Your executable(s) also depend on these dlls which are not included, you may or may not need to distribute them.

Make sure you have the license if you distribute any of them, and make sure you don't distribute files belonging to the operating system.

```

OLEAUT32.dll - C:\WINDOWS\system32\OLEAUT32.dll
USER32.dll - C:\WINDOWS\system32\USER32.dll
SHELL32.dll - C:\WINDOWS\system32\SHELL32.dll
ole32.dll - C:\WINDOWS\system32\ole32.dll
ADVAPI32.dll - C:\WINDOWS\system32\ADVAPI32.dll
VERSION.dll - C:\WINDOWS\system32\VERSION.dll
KERNEL32.dll - C:\WINDOWS\system32\KERNEL32.dll

```

运行完编译脚本以后会在当前文件夹下生成 dist 和 build 两个目录。其中 dist 目录中就是编译生成的文件。如果要在其他未安装 Python 的机器上运行编译好的程序，只要将 dist 目录复制到其他机器上即可。双击运行 MessageBox.exe，如图 10-4 所示。

在 setup.py 中除了导入必需的模块以外，只有一条语句。

```
distutils.core.setup(windows=['MessageBox.py'])
```

方括号中就是要编译的脚本名，前边的 windows 表示将其编译成 GUI 程序。如果要编译命令行界面的可执行文件，只要将 windows 改为 console。重新编译 MessageBox.py 后，双击运行，如图 10-5 所示。可以看到运行程序后多了一个命令行窗口。另外，如果需要将脚本编译成 Windows 服务，则可以使用 service 选项。



图 10-4 MessageBox.exe

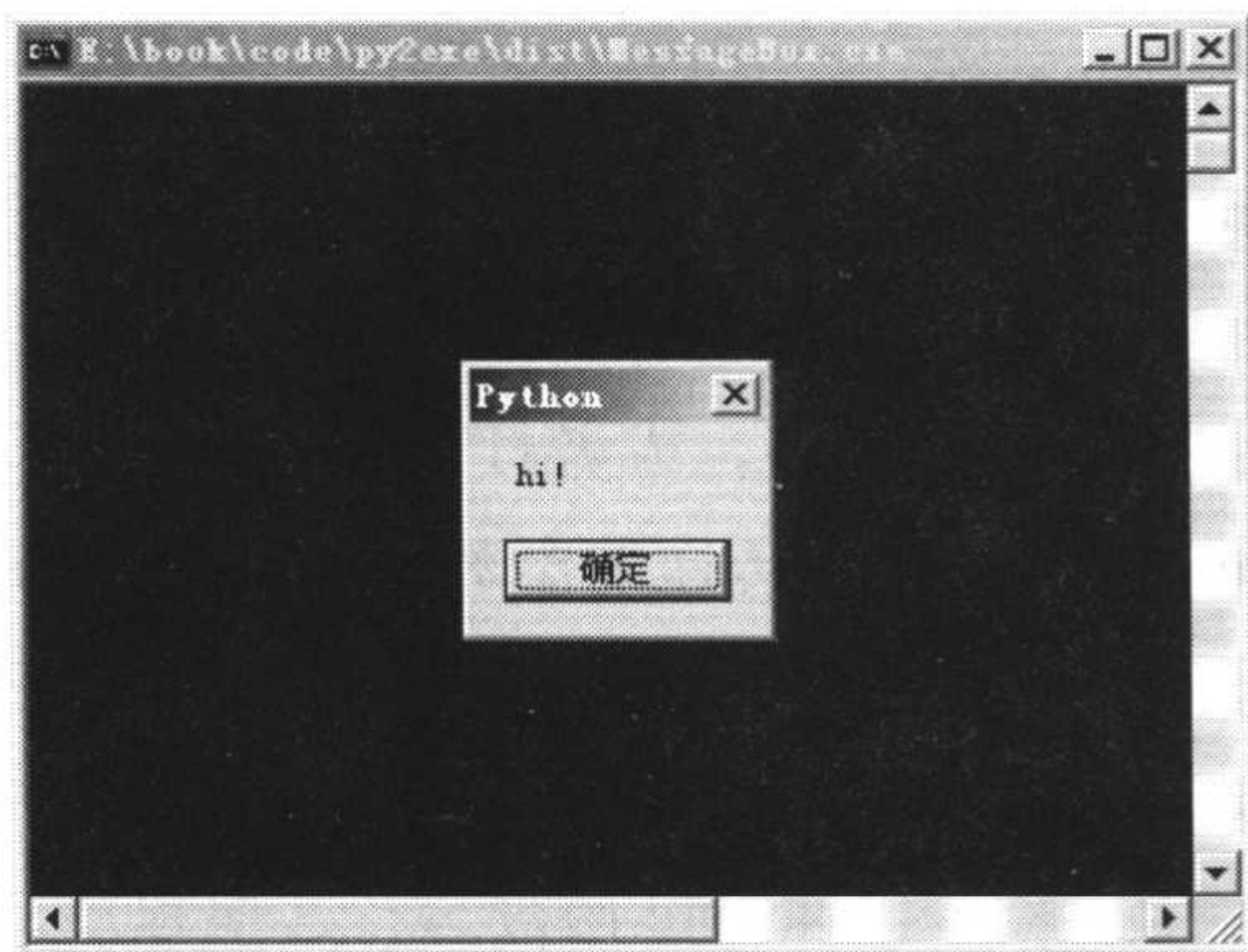


图 10-5 重新编译的 MessageBox.exe

10.4 运行其他程序

在 Python 中可以方便地使用 os 模块运行其他的脚本或者程序，这样就可以在脚本中直接使用其他脚本，或者程序提供的功能，而不必再次编写实现该功能的代码。为了更好地控制运行的进程，可以使用 win32process 模块中的函数。如果想进一步控制进程，则可以使用 ctype 模块，直接调用 kernel32.dll 中的函数。

10.4.1 使用 os.system 函数运行其他程序

os 模块中的 system() 函数可以方便地运行其他程序或者脚本。其函数原型如下所示。

```
os.system(command)
```

其参数含义如下所示。

- **command** 要执行的命令，相当于在 Windows 的 cmd 窗口中输入的命令。如果要向程序或者脚本传递参数，可以使用空格分隔程序及多个参数。

以下实例实现通过 os.system() 函数打开系统的记事本程序。

```
>>> import os
# 使用 os.system() 函数打开记事本程序
>>> os.system('notepad')
0 # 关闭记事本后的返回值
# 向记事本传递参数，打开 python.txt 文件
>>> os.system('notepad python.txt')
0
```

10.4.2 使用 ShellExecute 函数运行其他程序

除了使用 os 模块中的 os.system() 函数以外，还可以使用 win32api 模块中的 ShellExecute() 函数。其函数如下所示。

```
ShellExecute(hwnd, op, file, params, dir, bShow)
```

其参数含义如下所示。

- **hwnd**: 父窗口的句柄，如果没有父窗口，则为 0。
- **op**: 要进行的操作，为“open”、“print”或者为空。
- **file**: 要运行的程序，或者打开的脚本。
- **params**: 要向程序传递的参数，如果打开的为文件，则为空。
- **dir**: 程序初始化的目录。
- **bShow**: 是否显示窗口。

以下实例使用 ShellExecute 函数运行其他程序。

```
>>> import win32api
# 打开记事本程序，在后台运行，即显示记事本程序的窗口
```

```
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '', '', 0)
42
# 打开记事本程序, 在前台运行
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '', '', 1)
42
# 向记事本传递参数, 打开 python.txt
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', 'python.txt', '', 1)
42
# 在默认浏览器中打开 http://www.python.org 网站
>>> win32api.ShellExecute(0, 'open', 'http://www.python.org', '', '', 1)
42
# 在默认的媒体播放器中播放 E:\song.wma
>>> win32api.ShellExecute(0, 'open', 'E:\\song.wma', '', '', 1)
42
# 运行位于 E:\book\code 目录中的 MessageBox.py 脚本
>>> win32api.ShellExecute(0, 'open', 'E:\\book\\code\\MessageBox.py', '', '', 1)
42
```

可以看出, 使用 ShellExecute 函数, 就相当于在资源管理器中双击文件图标一样, 系统会打开相应的应用程序执行操作。

10.4.3 使用 CreateProcess 函数运行其他程序

为了便于控制通过脚本运行的程序, 可以使用 win32process 模块中的 CreateProcess() 函数。其函数原型如下所示。

```
CreateProcess(appName, commandLine, processAttributes, threadAttributes, bInheritHandles,
              dwCreationFlags, newEnvironment, currentDirectory, startupinfo)
```

其参数含义如下。

- appName: 可执行的文件名。
- commandLine: 命令行参数。
- processAttributes: 进程安全属性, 如果为 None, 则为默认的安全属性。
- threadAttributes: 线程安全属性, 如果为 None, 则为默认的安全属性。
- bInheritHandles: 继承标志。
- dwCreationFlags: 创建标志。
- newEnvironment: 创建进程的环境变量。
- currentDirectory: 进程的当前目录。
- startupinfo: 创建进程的属性。

以下实例使用 win32process.CreateProcess 函数运行记事本程序。

```
>>> import win32process
>>> win32process.CreateProcess('c:\\windows\\notepad.exe', '', None, None,
0, win32process.CREATE_NO_WINDOW, None, None, win32process.STARTUPINFO())
(<PyHANDLE:584>, <PyHANDLE:600>, 280, 3076) # 函数返回进程句柄、线程句柄、进程 ID, 以
```


及线程 ID

有了已创建进程的句柄就可以使用 `win32process.TerminateProcess` 函数结束进程, 或者使用 `win32event.WaitForSingleObject` 等待创建的线程结束。其函数原型分别如下。

```
TerminateProcess(handle, exitCode)
WaitForSingleObject(handle, milliseconds )
```

对于 `TerminateProcess` 参数含义分别如下。

- `handle`: 要操作的进程句柄。
- `exitCode`: 进程退出代码。

对于 `WaitForSingleObject` 参数含义分别如下。

- `handle`: 要操作的进程句柄。
- `milliseconds`: 等待的时间, 如果为-1, 则一直等待。

以下实例实现创建进程后并对其进行操作。

```
>>> import win32process
# 打开记事本程序, 获得其句柄
>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe', '', None,
None, 0, win32process.CREATE_NO_WINDOW, None, None, win32process.STARTUPINFO())
# 使用 TerminateProcess 函数终止记事本程序
>>> win32process.TerminateProcess(handle[0], 0)
# 导入 win32event 模块
>>> import win32event
# 创建进程获得句柄
>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe', '', None,
None, 0, win32process.CREATE_NO_WINDOW, None, None, win32process.STARTUPINFO())
# 等待进程结束
>>> win32event.WaitForSingleObject(handle[0], -1)
0 # 进程结束的返回值
```

10.4.4 使用 ctypes 调用 kernel32.dll 中的函数

使用 `ctypes` 模块可以使 Python 调用位于动态链接库中的函数。在 Python 2.5 版中已经包含了 `ctypes` 模块。如果使用其他版本的 Python, 可以到 <http://python.net/crew/theller/ctypes> 网站下载安装。`ctypes` 适用于 Python 2.3 版本及以上。

1. ctypes 简介

`ctypes` 为 Python 提供了调用动态链接库中函数的功能。使用 `ctypes` 可以方便地调用由 C 语言编写的动态链接库, 并向其传递参数。`ctypes` 定义了 C 语言中的基本数据类型, 并且可以实现 C 语言中的结构体和联合体。`ctypes` 可以工作在 Windows、Windows CE、Mac OS X、Linux、Solaris、FreeBSD、OpenBSD 等平台上, 基本上实现了跨平台。

以下的实例使用 `ctypes` 实现了在 Windows 下直接调用 `user32.dll` 中的 `MessageBoxA` 函数。运行后如图 10-6 所示。

```
>>> from ctypes import *
```

```
>>> user32 = windll.LoadLibrary('user32.dll') # 加载动态链接库
>>> user32.MessageBoxA(0, 'Ctypes is cool!', 'Ctypes', 0) # 调用 MessageBoxA 函数
1
```

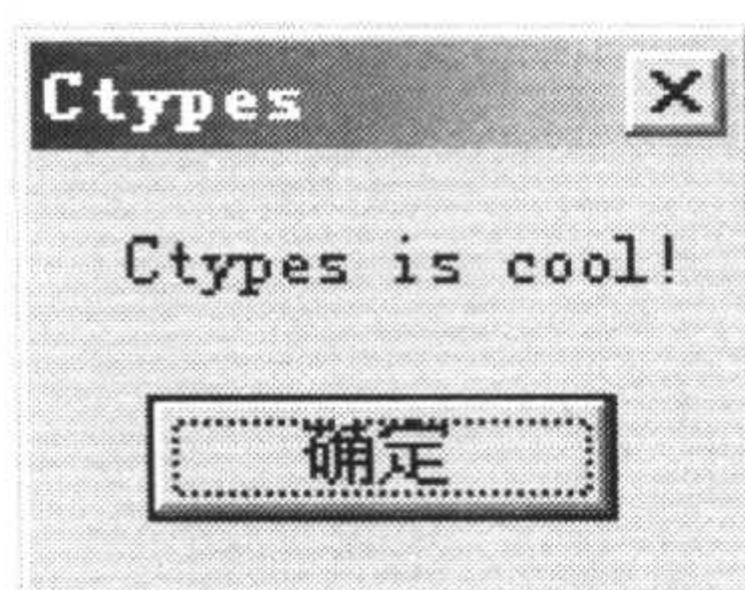


图 10-6 使用 ctypes

2. 数据类型与结构体

ctypes 实现 C 语言的基本数据类型，如表 10-2 所示列出了几个基本的数据类型的对照。

表 10-2 数据类型对照

ctypes 数据类型	C 数据类型	ctypes 数据类型	C 数据类型
c_char	char	c_float	float
c_short	short	c_double	double
c_int	int	c_void_p	void *
c_long	long		

在 Python 中要实现 C 语言的结构体，需要使用类。在 Python 中使用 ctypes 实现 Windows 中的 PROCESS_INFORMATION 结构体如下所示。

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION,
*LPPROCESS_INFORMATION;
```

在 Python 中由 ctypes 实现。

```
class _PROCESS_INFORMATION(Structure):
    _fields_ = [('hProcess', c_void_p),
                ('hThread', c_void_p),
                ('dwProcessId', c_ulong),
                ('dwThreadId', c_ulong)]
```

要声明一个 PROCESS_INFORMATION 类型的数据只要使用如下语句即可。

```
ProcessInfo = _PROCESS_INFORMATION()
```

如果在函数中要向结构体成员变量中赋值，可以使用 byref。byref 相当于 C 语言中的“&”。

3. 使用 kernel32.dll 中函数更改程序流程

在某些情况下，因为没有程序的源代码，但是又想让该程序在一定的情况下按照某一特

定的方式执行。此时就可以使用 WriteProcessMemory 函数，在创建程序进程后，修改其内存地址，按照要求执行。WriteProcessMemory 的函数原型如下所示。

```
BOOL WriteProcessMemory(
    HANDLE      hProcess,
    LPVOID      lpBaseAddress,
    LPCVOID     lpBuffer,
    SIZE_T      nSize,
    SIZE_T*     lpNumberOfBytesWritten
);
```

其参数含义如下。

- hProcess: 要写内存的进程句柄。
- lpBaseAddress: 要写的内存起始地址。
- lpBuffer: 写入值的地址。
- nSize: 写入值的大小。
- lpNumberOfBytesWritten: 实际写入的大小。

首先，在 Visual C++ 6.0 中创建一个示例程序。在 Visual C++中创建一个新的 Win32 Application，工程名为“ModifyMe”，如图 10-7 所示。

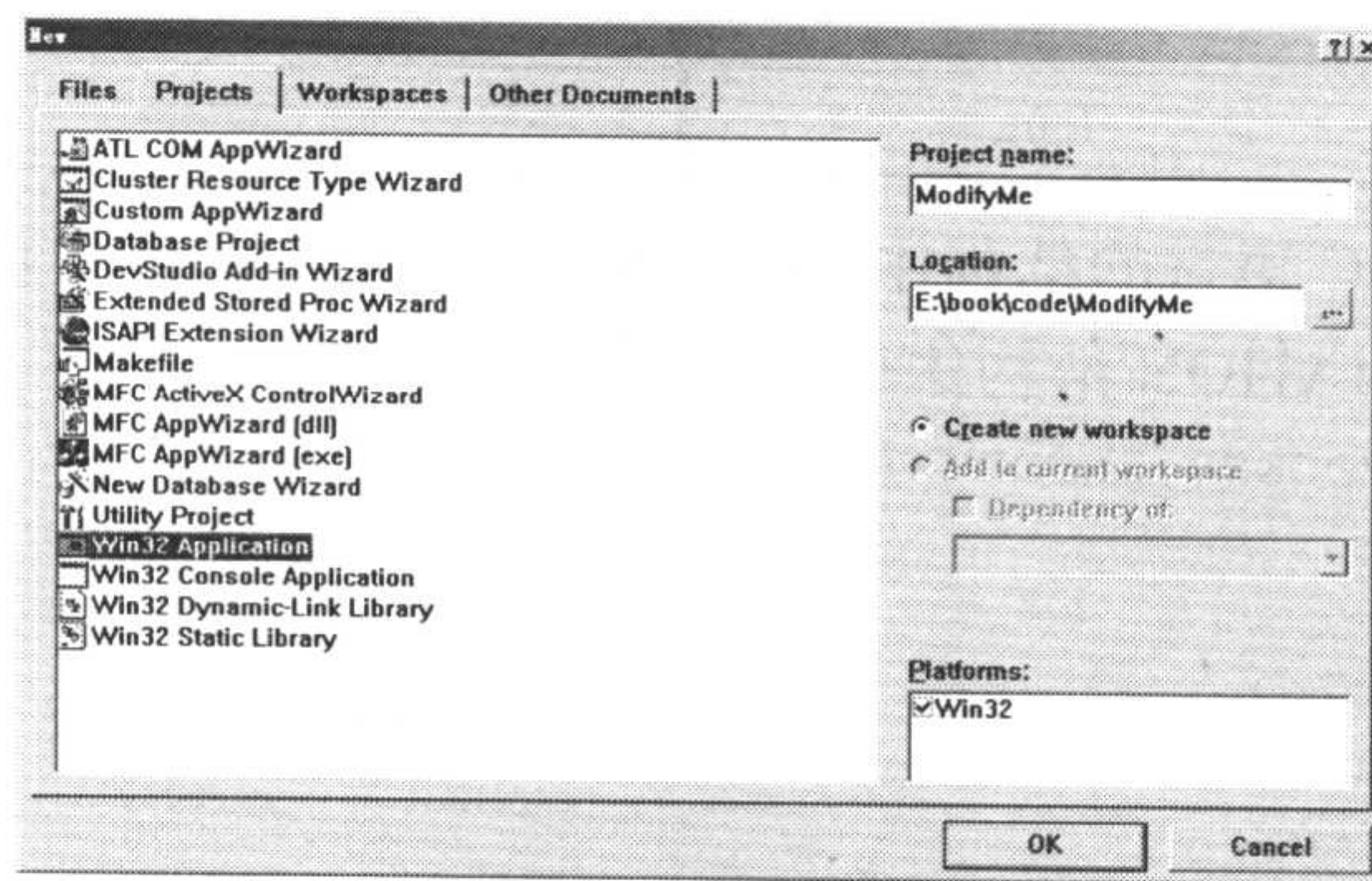


图 10-7 创建工程对话框

单击【OK】按钮，弹出如图 10-8 所示的对话框。单击【Finish】按钮后，会弹出一个确认对话框，单击【OK】按钮完成工程创建。新建一个 C/C++文件，将其命名为 ModifyMe.c，输入如下所示代码。编译 ModifyMe 后运行 ModifyMe.exe，如图 10-9 所示。

```
/* ModifyMe.c */
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    int a = 0;
    int b = 1;
    if ( a != b ) /* 此处即需要编写 Python 脚本修改的地方 */
```



```

{
    MessageBox(NULL, "Bad Python", "Python", MB_OK);
}
else
{
    MessageBox(NULL, "Good Python", "Python", MB_OK);
}
}

```

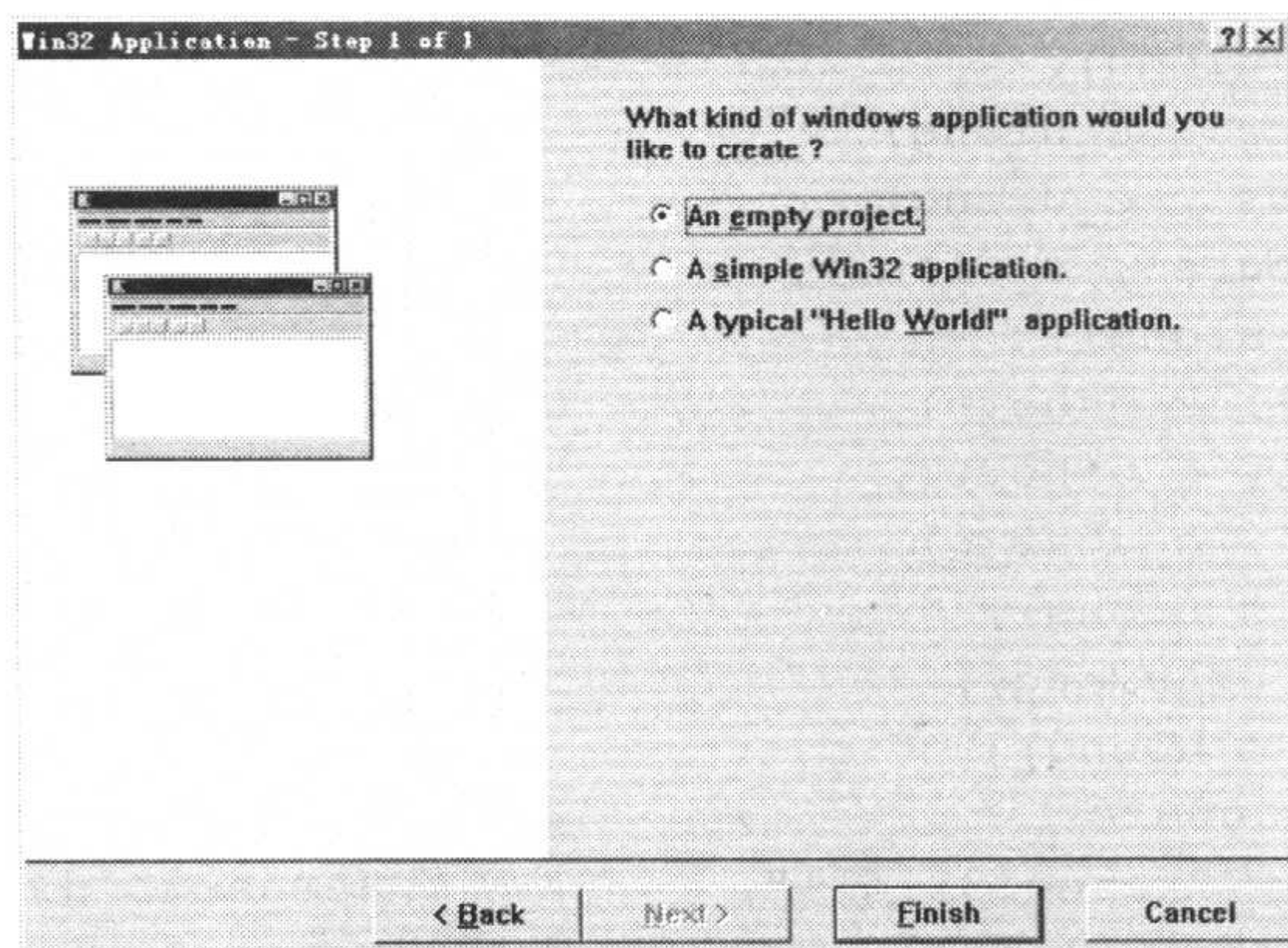


图 10-8 工程属性对话框

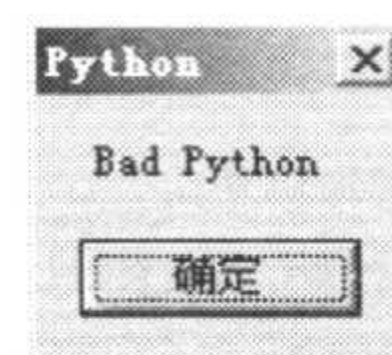


图 10-9 修改前的 ModifyMe 程序

为找到“if (a != b)”的反汇编后的代码，需要在 ModifyMe.c 中设置断点，进入调试模式，查看汇编代码，如下所示。可以看出，关键是位于地址 0040103C 处的 je 指令。

```

7:      if ( a != b )
00401036  mov     eax,dword ptr [ebp-4]
00401039  cmp     eax,dword ptr [ebp-8]
0040103C  je      WinMain+4Dh (0040105d)

```

0040103C 处的 je 指令表示如果 a 与 b 的值相等，则程序跳转至 0040105d 处执行。而程序中 a 与 b 的值并不相等，因此程序没有跳转。这里需要将 je 指令改为 jne。其中 je 指令反汇编后的十六进制值为 0x74，而 jne 则为 0x75。如果要修改程序流程，只要向 0040103C 地址处写入一个字节，将 je 改为 jne，即向 0040103C 处写入 0x75。编写的修改脚本代码如下所示。

```

# -*- coding:utf-8 -*-
# file: ModifyMemory.py
#
from ctypes import *
# 定义 _PROCESS_INFORMATION 结构体
class _PROCESS_INFORMATION(Structure):
    _fields_ = [('hProcess', c_void_p),
                ('hThread', c_void_p),
                ('dwProcessId', c_ulong),
                ('dwThreadId', c_ulong)]

```

```

# 定义 _STARTUPINFO 结构体
class _STARTUPINFO(Structure):
    _fields_ = [('cb', c_ulong),
                 ('lpReserved', c_char_p),
                 ('lpDesktop', c_char_p),
                 ('lpTitle', c_char_p),
                 ('dwX', c_ulong),
                 ('dwY', c_ulong),
                 ('dwXSize', c_ulong),
                 ('dwYSize', c_ulong),
                 ('dwXCountChars', c_ulong),
                 ('dwYCountChars', c_ulong),
                 ('dwFillAttribute', c_ulong),
                 ('dwFlags', c_ulong),
                 ('wShowWindow', c_ushort),
                 ('cbReserved2', c_ushort),
                 ('lpReserved2', c_char_p),
                 ('hStdInput', c_ulong),
                 ('hStdOutput', c_ulong),
                 ('hStdError', c_ulong)]

NORMAL_PRIORITY_CLASS = 0x00000020
kernel32 = windll.LoadLibrary("kernel32.dll")
CreateProcess = kernel32.CreateProcessA
ReadProcessMemory = kernel32.ReadProcessMemory
WriteProcessMemory = kernel32.WriteProcessMemory
TerminateProcess = kernel32.TerminateProcess

# 声明结构体
ProcessInfo = _PROCESS_INFORMATION()
StartupInfo = _STARTUPINFO()
file = 'ModifyMe.exe'
address = 0x0040103c
buffer = c_char_p("_")
bytesRead = c_ulong(0)
bufferSize = len(buffer.value)

# 创建进程
if CreateProcess(file, 0, 0, 0, 0, NORMAL_PRIORITY_CLASS, 0, 0, byref(StartupInfo),
byref(ProcessInfo)):
    # 读取要修改的内存地址，以判断是否是要修改的文件
    if ReadProcessMemory(ProcessInfo.hProcess, address, buffer, bufferSize,
byref(bytesRead)):
        if buffer.value == '\x74':
            buffer.value = '\x75'
            # 修改内存
            if WriteProcessMemory(ProcessInfo.hProcess, address, buffer, bufferSize,
byref(bytesRead)):
                print '成功改写内存!'
            else:
                print '写内存错误!'
        else:
            # 定义 NORMAL_PRIORITY_CLASS
            # 加载 kernel32.dll
            # 获得 CreateProcess 函数地址
            # 获得 ReadProcessMemory 函数地址
            # 获得 WriteProcessMemory 函数地址
            # 要进行修改的文件
            # 要修改的内存地址
            # 缓冲区地址
            # 读入的字节数
            # 缓冲区大小
            # 修改缓冲区内的值，将其写入内存

```

```
print '打开了错误的文件!'
TerminateProcess(ProcessInfo.hProcess, 0) # 如果不是要修改的文件, 则终止进程
else:
    print '读内存错误!'
else:
    print '不能创建进程!'
```

运行脚本后, 如图 10-10 所示。



图 10-10 修改后的 ModifyMe

第 11 章 使用 PythonWin 编写 GUI

在前边的章节中已经为 Python 扩展调用 MFC 创建了一个对话框。实际上, PythonWin 也是通过扩展的形式对 MFC 的函数进行封装的。通过使用 PythonWin 中的 win32gui 和 win32ui 模块可以调用 Windows API, 或者使用 MFC 来创建 GUI 界面。

11.1 Windows GUI 编程概述

在 Windows 操作系统下, 可以直接使用 Windows API 创建 GUI 程序, 但使用 Windows API 较为繁琐。Windows 提供了 MFC 类库对 Windows API 进行封装。使用 MFC 同样可以创建 GUI 程序, 但 MFC 类库较为庞大, 不容易掌握。

11.1.1 使用 Windows API 创建窗口

在 Python 中使用 PythonWin 提供的 Windows API, 与在 VC++ 6.0 中使用 Windows API 编写 GUI 的过程一样。

1. 创建窗口

使用 Windows API 创建窗口, 首先注册窗口类、定义消息回调函数, 然后创建并显示窗口。以下是 WinGUI.py 脚本, 它使用 PythonWin 提供的 Windows API 创建一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: WinGUI.py
#
import win32gui
from win32con import *
# 窗口消息处理函数
def WndProc(hwnd, msg, wParam, lParam):
    if msg == WM_PAINT:
        hdc, ps = win32gui.BeginPaint(hwnd)
        rect = win32gui.GetClientRect(hwnd)
        win32gui.DrawText(hdc,
                           'GUI Python',
                           len('GUI Python'),
                           rect,
```

```

        DT_SINGLELINE|DT_CENTER|DT_VCENTER)
    win32gui.EndPaint(hwnd, ps)
    if msg == WM_DESTROY:
        win32gui.PostQuitMessage(0)
    return win32gui.DefWindowProc(hwnd, msg, wParam, lParam)
# 生成窗口类, 并对相关项赋值
wc = win32gui.WNDCLASS()
wc.hbrBackground = COLOR_BTNFACE + 1
wc.hCursor = win32gui.LoadCursor(0, IDC_ARROW)
wc.hIcon = win32gui.LoadIcon(0, IDI_APPLICATION)
wc.lpszClassName = "Python on Windows"
wc.lpfnWndProc = WndProc
# 注册窗口类
reg = win32gui.RegisterClass(wc)
# 创建窗口
hwnd = win32gui.CreateWindow(
    reg, 'Python', WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT,
    0, 0, 0, None)
# 显示窗口
win32gui.ShowWindow(hwnd, SW_SHOWNORMAL)
win32gui.UpdateWindow(hwnd)
# 进入消息循环, 直至窗口结束
win32gui.PumpMessages()

```

运行 WinGUI.py 脚本后会创建如图 11-1 所示的窗口。

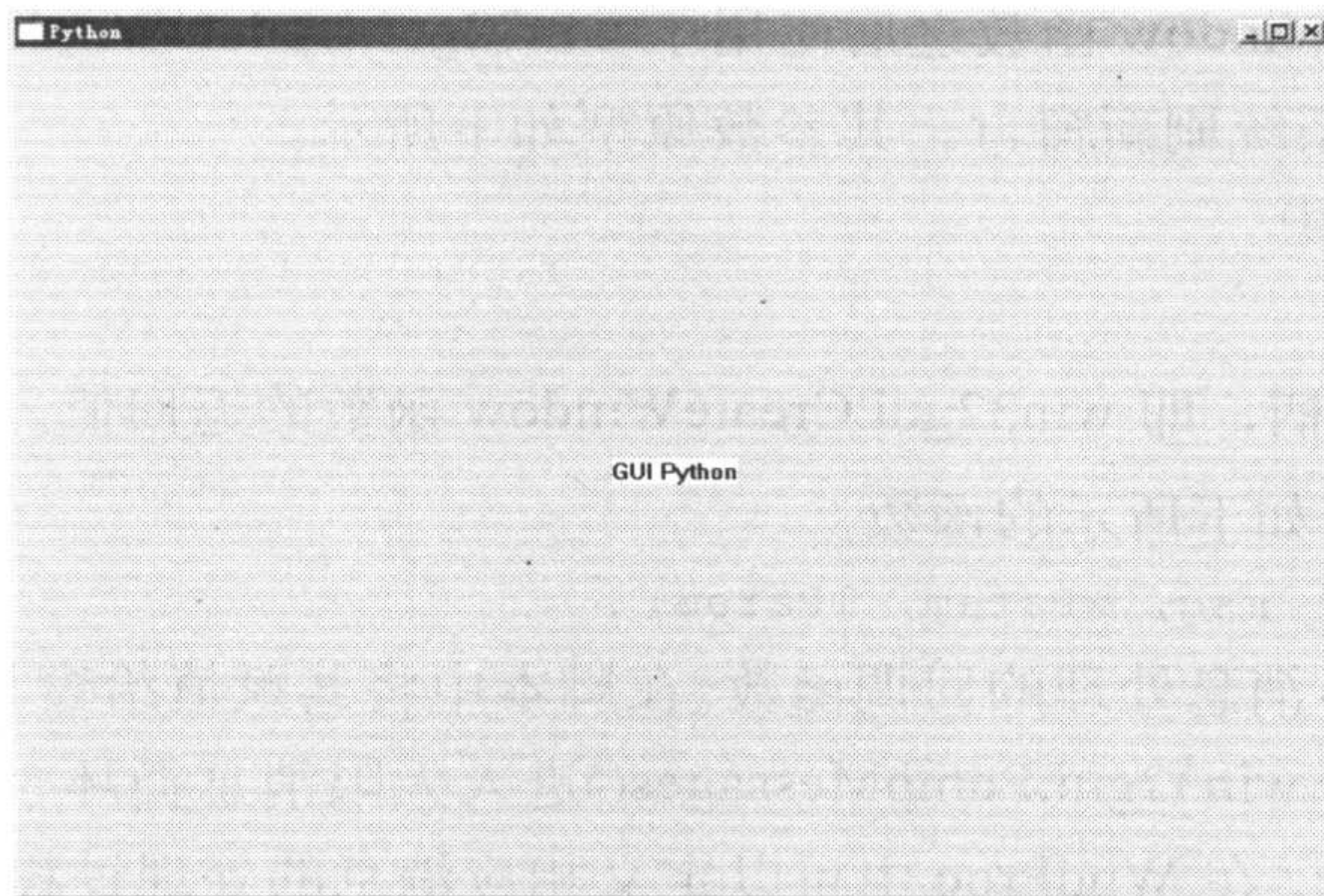


图 11-1 Python GUI 窗口

2. 脚本说明

在脚本中主要使用了创建窗口、显示窗口、输出文字等函数。其中函数 `win32gui.CreateWindow` 用于创建窗口, 它返回创建窗口的句柄。其参数原型如下所示。

```
CreateWindow(className, windowTitle, style, x, y, width, height, parent, menu, hinstance, reserved)
```

其参数含义如下。

- className: 注册的窗口类名。
- windowTitle: 窗口标题名。
- style: 窗口样式。
- x: 窗口左上角 x 坐标。
- y: 窗口左上角 y 坐标。
- width: 窗口的宽度。
- height: 窗口的高度。
- parent: 父窗口句柄, 如果没有, 则设置为 0。
- menu: 菜单 ID, 如果没有, 则设置为 0。
- hinstance: 该值在 Windows XP 下被忽略, 设置为 0。
- reserved: 保留, 应为 None。

当使用 win32gui.CreateWindow 创建窗口后, 还要使用 win32gui.ShowWindow 函数才能显示窗口。其参数原型如下所示。

```
ShowWindow(hwnd, cmdShow)
```

其参数含义如下。

- hwnd: 窗口句柄, 即 win32gui.CreateWindow 函数的返回值。
- cmdShow: 指定窗口的显示状态。

在 win32gui.ShowWindow 函数之后使用了 win32gui.UpdateWindow 函数, 其作用是发送 WM_PAINT 消息通知系统刷新窗口。其参数原型如下所示。

```
UpdateWindow(hwnd)
```

其参数含义如下。

- hwnd: 窗口句柄, 即 win32gui.CreateWindow 函数的返回值。

在脚本中还定义了如下所示的函数。

```
def WndProc(hwnd, msg, wParam, lParam)
```

该函数即 Windows 消息处理的回调函数。在脚本中将其赋值给窗口类的 wc.lpfnWndProc。当窗口创建以后, 使用 win32gui.PumpMessages() 进入无限消息循环, 处理窗口消息。窗口消息首先传递给 WndProc, 在 WndProc 中可以定义相应消息的处理过程。在 WinGUI.py 中处理了 WM_PAINT 和 WM_DESTROY 消息。其中在 WM_PAINT 消息中向窗口输出“GUI Python”。在 WM_DESTROY 消息中向窗口发送退出消息。对于其他的消息, 则调用默认的消息处理函数 win32gui.DefWindowProc。

11.1.2 使用 MFC 创建窗口

MFC 对基本的 Windows API 函数进行了封装。使用 MFC 创建窗口的大致过程是: 首先, 创

建窗口类；然后，重载消息处理方法；最后，类实例化，创建窗口。当窗口创建后，要调用 `RunModalLoop` 方法进入窗口消息循环，否则窗口将立刻关闭。另外需要重载 `OnClose` 方法，使用 `EndModalLoop` 方法结束消息循环。如下所示的 `MFCGUI.py` 脚本为 MFC 创建了一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: MFCGUI.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def _init_(self):
        window.Wnd._init_(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
            win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
            'MFC GUI', WS_OVERLAPPEDWINDOW, \
            (100, 100, 400, 300), None, 0, None)
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    # 重载 OnPaint 方法，在窗口中输出“MFC GUI”
    def OnPaint(self):
        dc, ps = self.BeginPaint()
        dc.DrawText('MFC GUI',
            self.GetClientRect(),
            DT_SINGLELINE | DT_CENTER | DT_VCENTER)
        self.EndPaint(ps)
w = MyWnd()
w.ShowWindow()
w.UpdateWindow()
w.RunModalLoop(1)
```

生成窗口对象
显示窗口
刷新窗口
进入消息循环

运行 `MFCGUI.py` 脚本后将创建如图 11-2 所示的窗口。

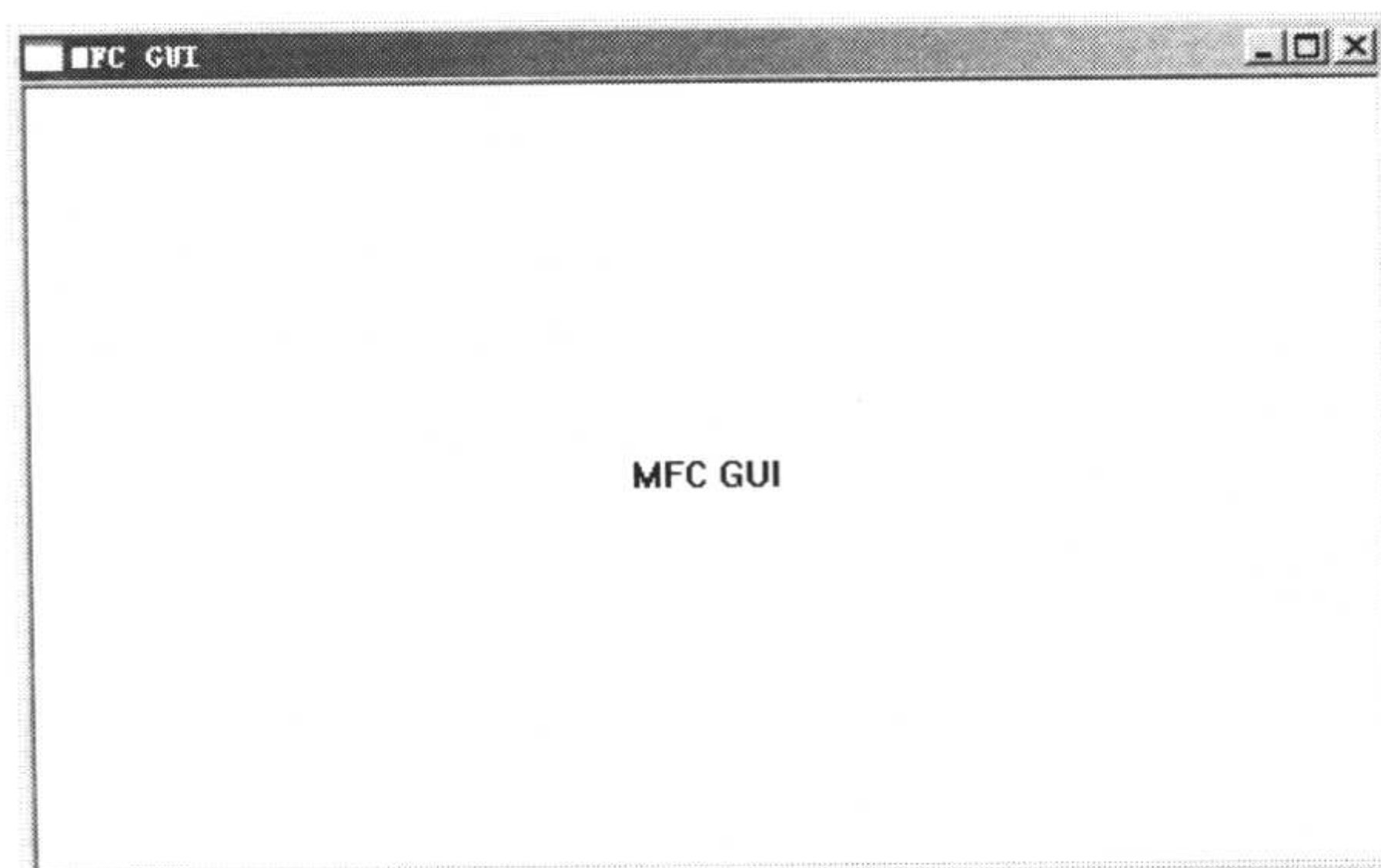


图 11-2 使用 MFC 创建 GUI 窗口

11.2 对话框

在多数 GUI 程序中都会使用到对话框。对话框用来和用户交互完成与程序相关选项的配置。对于某些简单的 GUI 程序，其本身就是一个对话框。创建对话框要比创建窗口容易。如果不想脚本太复杂，可以考虑在脚本中使用对话框替代窗口。

11.2.1 创建对话框

通过重载 pywin.mfc 模块中的 Dialog 可以创建简单的对话框。对话框的创建过程与窗口的创建过程类似，但相对要简单些。如下所示的 SimpleDialog.py 脚本，通过重载 pywin.mfc 模块中的 dialog.Dialog 创建了一个简单的对话框。

```
# -*- coding:utf-8 -*-
# file: SimpleDialog.py
#
import win32ui                                     # 导入 Win32ui 模块
import win32con                                     # 导入 win32con 模块
from pywin.mfc import dialog                       # 导入 pywin.mfc 中的 dialog 模块
class MyDialog(dialog.Dialog):                     # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                         # 重载初始化函数
        dialog.Dialog.OnInitDialog(self)           # 调用父类的初始化函数
style = (win32con.DS_MODALFRAME |                 # 定义对话框样式
        win32con.WS_POPUP |
        win32con.WS_VISIBLE |
        win32con.WS_CAPTION |
        win32con.WS_SYSMENU |
        win32con.DS_SETFONT)
di = ['Python',                                    # 设置对话框属性，设置标题为 "Python"
      (0,0,300,180),                               # 设置对话框位置及大小
      style,                                         # 设置对话框样式
      None,                                         # 设置对话框扩展样式
      (8, "MS Sans Serif")]                         # 设置对话框字体及大小
init = []                                           # 定义对话框初始化信息列表
init.append(di)
mydialog = MyDialog(init)                          # 生成对话框实例对象
mydialog.DoModal()                                # 显示对话框
```

运行 SimpleDialog.py 脚本后创建如图 11-3 所示的对话框。

11.2.2 向对话框中添加控件

在上一小节的例子中，只创建出一个简单的对话框。一个较完整的对话框应包含按钮、文本框等控件。使用 pywin.mfc 模块中的 Dialog 模块创建对话框时，可以将控件信息放在初始化参数中。当对话框创建时将创建相应的控件。

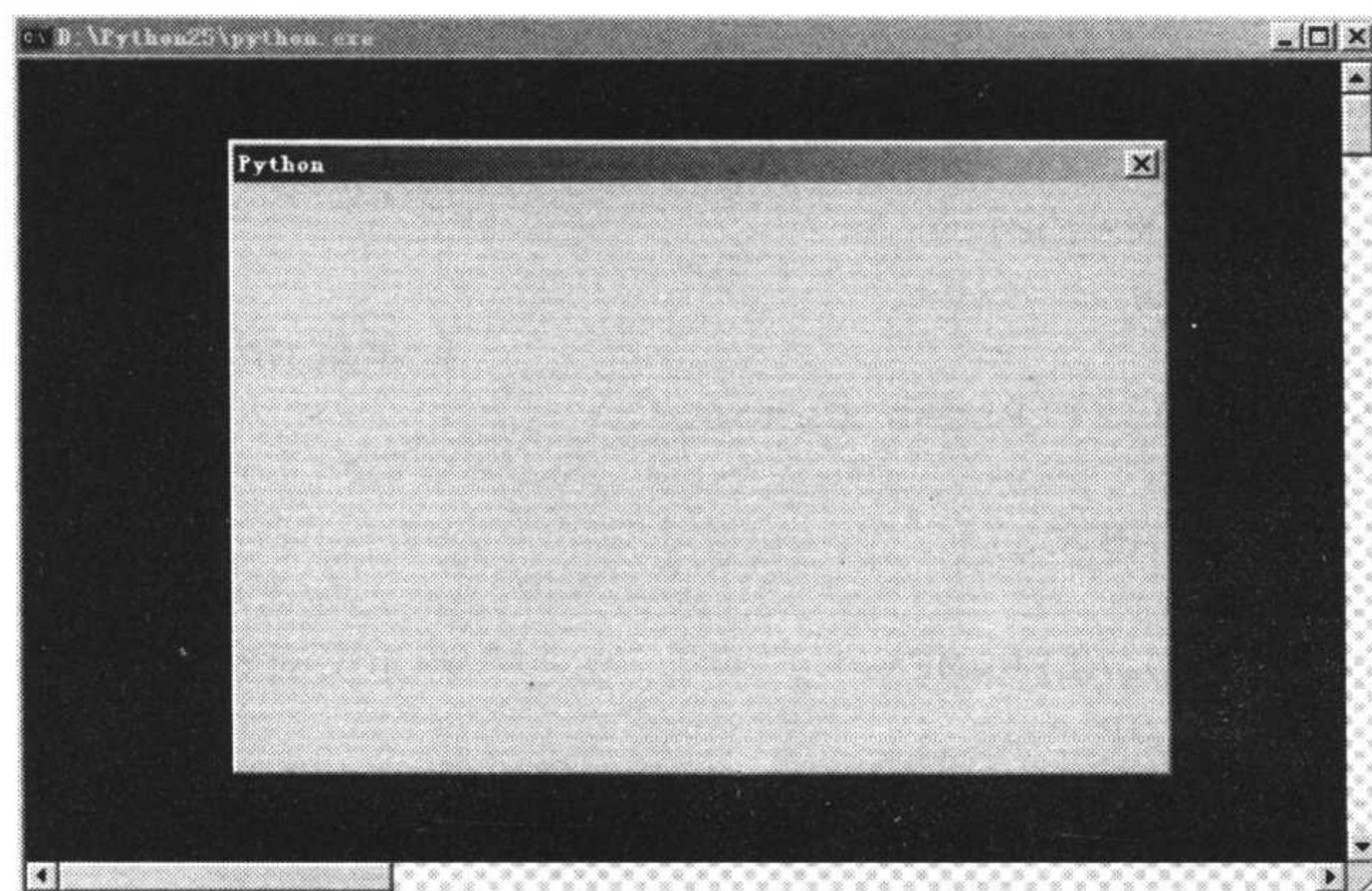


图 11-3 一个简单的对话框

使用 PythonWin 创建控件时，需要使用一个列表来保存控件的信息。其由以下几项组成。

- 控件类型，可以为字符串或者数字。例如文本框、按钮等。
- 控件标题。
- 控件 ID。
- 控件的位置，为包含 4 项的元组。其分别为居于对话框左上角的 x 坐标、y 坐标、宽度以及高度。
- 控件样式。

其中控件类型如表 11-1 所示。

表 11-1

控件类型

控 件 名 称	字 符 表 示	数 字 表 示	控 件 名 称	字 符 表 示	数 字 表 示
按钮	Button	128	列表框	ListBox	131
组合框	ComboBox	133	滚动条	ScrollBar	132
文本框	Edit	129	标签	Static	130

如下所示的 Dialog.py 脚本创建了一个对话框，并向对话框中添加了文本框、按钮、标签等控件。

```
# -*- coding:utf-8 -*-
# file: Dialog.py
#
import win32ui
import win32con
from pywin.mfc import dialog
class MyDialog(dialog.Dialog):
    def OnInitDialog(self):
        dialog.Dialog.OnInitDialog(self)
```

```
# 导入 win32ui 模块
# 导入 win32con 模块
# 从 pywin.mfc 中导入 dialog
# 通过继承 dialog.Dialog 生成对话框类
# 重载对话框初始化方法
# 调用父类的对话框初始化方法
```



```

def OnOK(self):
    win32ui.MessageBox('Press Ok', \
        'Python', \
        win32con.MB_OK)
    self.EndDialog(1)
def OnCancel(self):
    win32ui.MessageBox('Press Cancel', \
        'Python', \
        win32con.MB_OK)
    self.EndDialog(1)
style = (win32con.DS_MODALFRAME |
    win32con.WS_POPUP |
    win32con.WS_VISIBLE |
    win32con.WS_CAPTION |
    win32con.WS_SYSMENU |
    win32con.DS_SETFONT)
childstyle = (win32con.WS_CHILD |
    win32con.WS_VISIBLE)
buttonstyle = win32con.WS_TABSTOP | childstyle
di = ['Python',
    (0,0,300,180),
    style,
    None,
    (8, "MS Sans Serif")]
ButOK = (['Button',
    "OK",
    win32con.IDOK,
    (80,150, 50, 14),
    buttonstyle | win32con.BS_PUSHBUTTON])
ButCancel = (['Button',
    "Cancel",
    win32con.IDCANCEL,
    (160, 150, 50, 14),
    buttonstyle | win32con.BS_PUSHBUTTON])
Stadic = (['Static',
    'Python Dialog',
    12,
    (130,50,60,14),
    childstyle])
Edit = (['Edit',
    '',
    13,
    (130,80,60,14),
    childstyle|win32con.ES_LEFT|
    win32con.WS_BORDER|win32con.WS_TABSTOP])
init = []
init.append(di)
init.append(ButOK)
init.append(ButCancel)
init.append(Stadic)

```

重载 OnOK 方法

重载 OnCancel 方法

定义对话框样式

定义控件样式

定义按钮样式

设置对话框属性

设置 OK 按钮属性

设置 Cancel 按钮属性

设置标签属性

设置文本框属性

初始化信息列表

```

init.append(Edit)
mydialog = MyDialog(init)           # 生成对话框实例对象
mydialog.DoModal()                  # 创建对话框

```

运行 Dialog.py 脚本后将创建如图 11-4 所示的对话框。

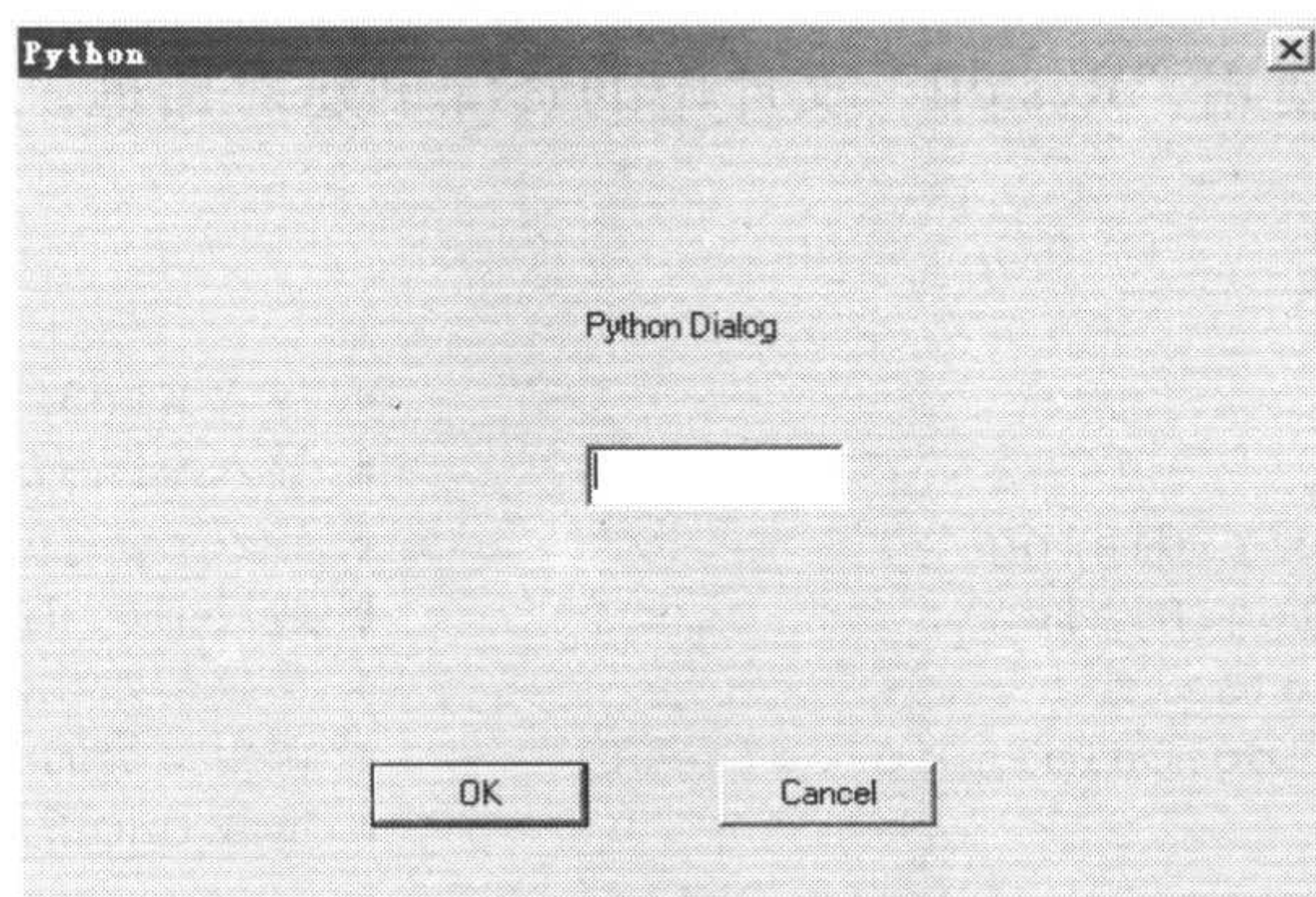


图 11-4 向对话框中添加控件

11.2.3 使用 DLL 文件中的资源

在 Python 中直接定义对话框中的控件比较繁琐。实际上，win32ui 模块中的 LoadDialogResource 函数可以载入 DLL 文件中的对话框资源，这样只要在 VC++ 6.0，或者 Visual Studio 2005 中创建一个 DLL 工程，向其添加对话框及所需控件，然后就可以直接载入 DLL，通过 win32ui.LoadDialogResource 函数载入其资源，在 Python 中使用其中的对话框及控件等。

在使用 DLL 中的资源之前要使用 win32ui.LoadLibrary 函数载入 DLL 文件。其函数原型如下所示。

```
LoadLibrary(fileName)
```

其参数含义如下。

- filename: 要载入的 DLL 文件名。

当 win32ui.LoadLibrary 函数正确载入 DLL 文件后将返回一个 PyDLL 类型的值。然后就可以通过该值以及对话框的 ID 使用 win32ui.LoadDialogResource 函数载入 DLL 中的对话框。win32ui.LoadDialogResource 函数原型如下所示。

```
LoadDialogResource(idRes, dll)
```

其参数含义如下。

- idRes: 对话框的 ID。
- dll: 使用 win32ui.LoadLibrary 载入 DLL 文件的返回值。

win32ui.LoadDialogResource 函数返回一个资源列表，然后在对话框类的实例化时，调用该列表就可以使用 DLL 文件中的对话框。首先在 VC++ 6.0 中创建一个名为“Res”的“MFC AppWizard(dll)”工程，按照默认选项创建工程。然后向工程中添加对话框及相应的资源，完

成后编译 DLL。最后,需要打开工程中的“Resource.h”文件查看对话框的 ID。在“Resource.h”会找到类似如下所示的语句。

```
#define IDD_DIALOG1 7000
```

其中 7000 就是对话框的 ID,也就是 LoadDialogResource 函数中的 idRes 参数。如下所示的 DllRes.py 脚本通过使用“Res.dll”文件中的资源创建对话框。

```
# -*- coding:utf-8 -*-
# file: DllRes.py
#
import win32ui                                     # 导入 Win32ui 模块
import win32con                                     # 导入 win32con 模块
from pywin.mfc import dialog                       # 导入 pywin.mfc 中的 dialog 模块
class MyDialog(dialog.Dialog):                     # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                        # 重载初始化函数
        dialog.Dialog.OnInitDialog(self)          # 调用父类的初始化函数
    def OnOK(self):                                # 重载 OnOK 方法
        win32ui.MessageBox('Press Ok', \
                             'Python', \
                             win32con.MB_OK)
        self.EndDialog(1)
    def OnCancel(self):                            # 重载 OnCancel 方法
        win32ui.MessageBox('Press Cancel', \
                             'Python', \
                             win32con.MB_OK)
        self.EndDialog(1)
dll = win32ui.LoadLibrary('Res.dll')               # 载入 DLL 文件
l = win32ui.LoadDialogResource(7000,dll)           # 载入 DLL 文件中的对话框
mydialog = MyDialog(l)                             # 生成对话框实例对象
mydialog.DoModal()                                 # 显示对话框
```

运行 DllRes.py 脚本后将创建如图 11-5 所示的对话框。

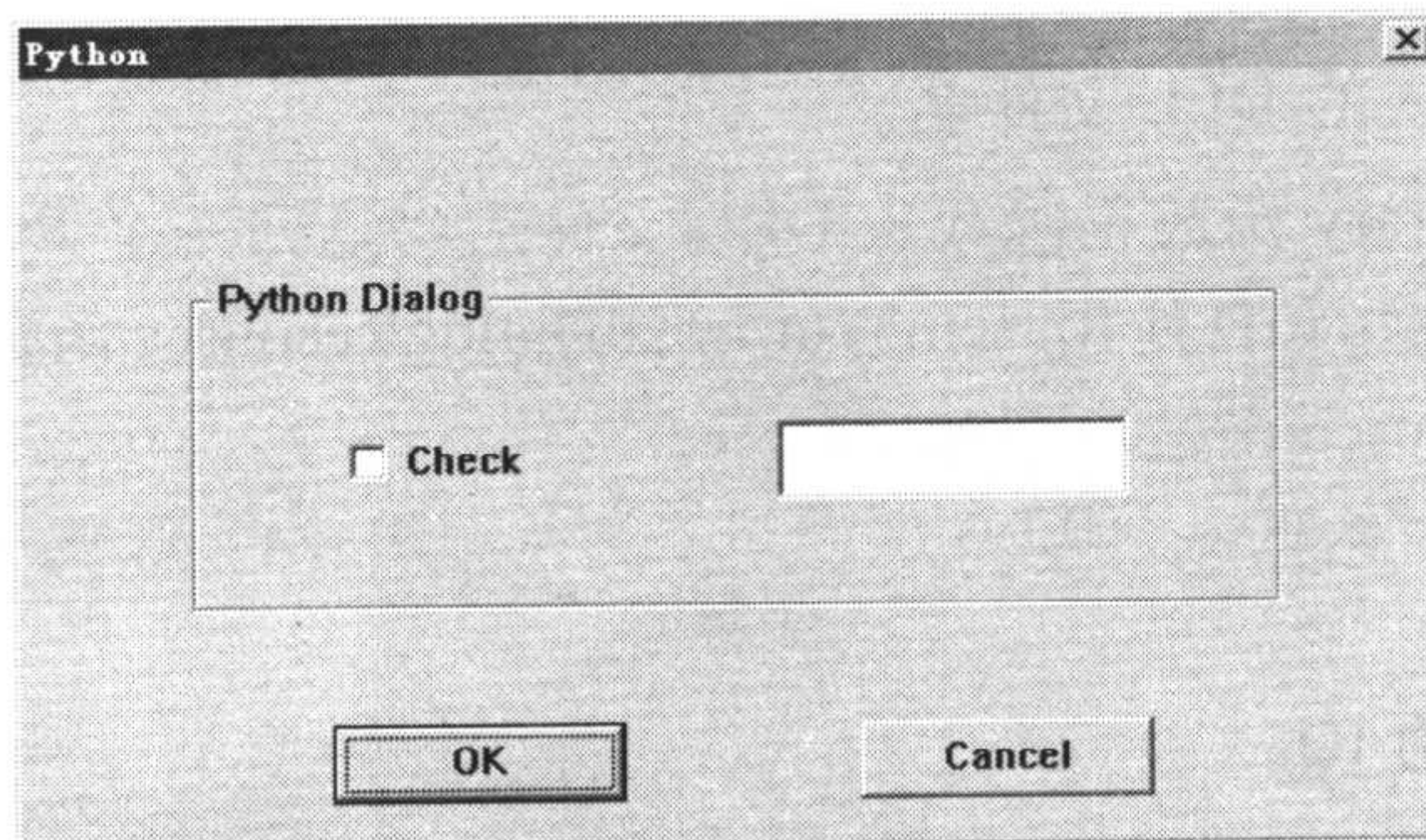


图 11-5 使用 DLL 中的资源创建对话框

11.2.4 处理按钮消息

在前面的例子中通过重载了父类的 OnOK 方法处理 OK 按钮单击的消息。如果添加其

他的按钮，则需要使用 HookCommand 方法，为按钮设置相应的消息处理方法。其原型如下所示。

```
HookCommand(method, id)
```

其参数含义如下。

- method: 处理消息的方法。
- id: 按钮的 ID。

其中按钮消息处理的方法应定义为如下所示的形式。

```
def OnButton(self, lParam, wParam):                                     # 处理 Button 单击消息
    <消息处理语句>
```

除了必需的 self 参数以外，还应该有 wParam 和 lParam 参数。其中 wParam 参数的值总为 0，而 lParam 参数的值则为所单击按钮的 ID。如下所示的 DialogCmd.py 脚本使用 HookCommand 方法处理按钮消息。

```
# -*- coding:utf-8 -*-
# file: DialogCmd.py
#
import win32ui                                                         # 导入 win32ui 模块
import win32con                                                         # 导入 win32con 模块
from pywin.mfc import dialog                                           # 从 pywin.mfc 中导入 dialog
class MyDialog(dialog.Dialog):                                         # 通过继承 dialog.Dialog 生成对话框类
    def OnInitDialog(self):                                           # 重载对话框初始化方法
        dialog.Dialog.OnInitDialog(self)                             # 调用父类的对话框初始化方法
        self.HookCommand(self.OnButton1, 1051)                       # 设置按钮消息处理方法
        self.HookCommand(self.OnButton2, 1052)
    def OnButton1(self, wParam, lParam):                               # 处理 Button1 单击消息
        win32ui.MessageBox('Button1', \
                             'Python', \
                             win32con.MB_OK)
        self.EndDialog(1)
    def OnButton2(self, wParam, lParam):                               # 处理 Button2 单击消息
        text = self.GetDlgItemText(1054)                             # 获得文本框中的文字
        win32ui.MessageBox(text, \
                             'Python', \
                             win32con.MB_OK)
        self.EndDialog(1)
style = (win32con.DS_MODALFRAME |                                     # 定义对话框样式
         win32con.WS_POPUP |
         win32con.WS_VISIBLE |
         win32con.WS_CAPTION |
         win32con.WS_SYSMENU |
         win32con.DS_SETFONT)
childstyle = (win32con.WS_CHILD |                                    # 定义控件样式
              win32con.WS_VISIBLE)
buttonstyle = win32con.WS_TABSTOP | childstyle                      # 定义按钮样式
di = ['Python',                                                       # 设置对话框属性
      (0, 0, 300, 180),
```

```

        style,
        None,
        (8, "MS Sans Serif"))
Button1 = (['Button',
            'Button1',
            1051,
            (80,150, 50, 14),
            buttonstyle | win32con.BS_PUSHBUTTON])
Button2 = (['Button',
            'Button2',
            1052,
            (160, 150, 50, 14),
            buttonstyle | win32con.BS_PUSHBUTTON])
Stadic = (['Static',
            'Python Dialog',
            1053,
            (130,50,60,14),
            childstyle])
Edit = (['Edit',
        '',
        1054,
        (130,80,60,14),
        childstyle|win32con.ES_LEFT|
        win32con.WS_BORDER|win32con.WS_TABSTOP])
init = []
init.append(di)
init.append(Button1)
init.append(Button2)
init.append(Stadic)
init.append(Edit)
mydialog = MyDialog(init)
mydialog.DoModal()

```

设置 Button1 按钮属性

设置 Button2 按钮属性

设置标签属性

设置文本框属性

初始化信息列表

生成对话框实例对象

创建对话框

运行 DialogCmd.py 脚本后，单击 Button1 按钮时如图 11-6 所示。在文本框中输入“Python!”后，单击 Button2 按钮时如图 11-7 所示。

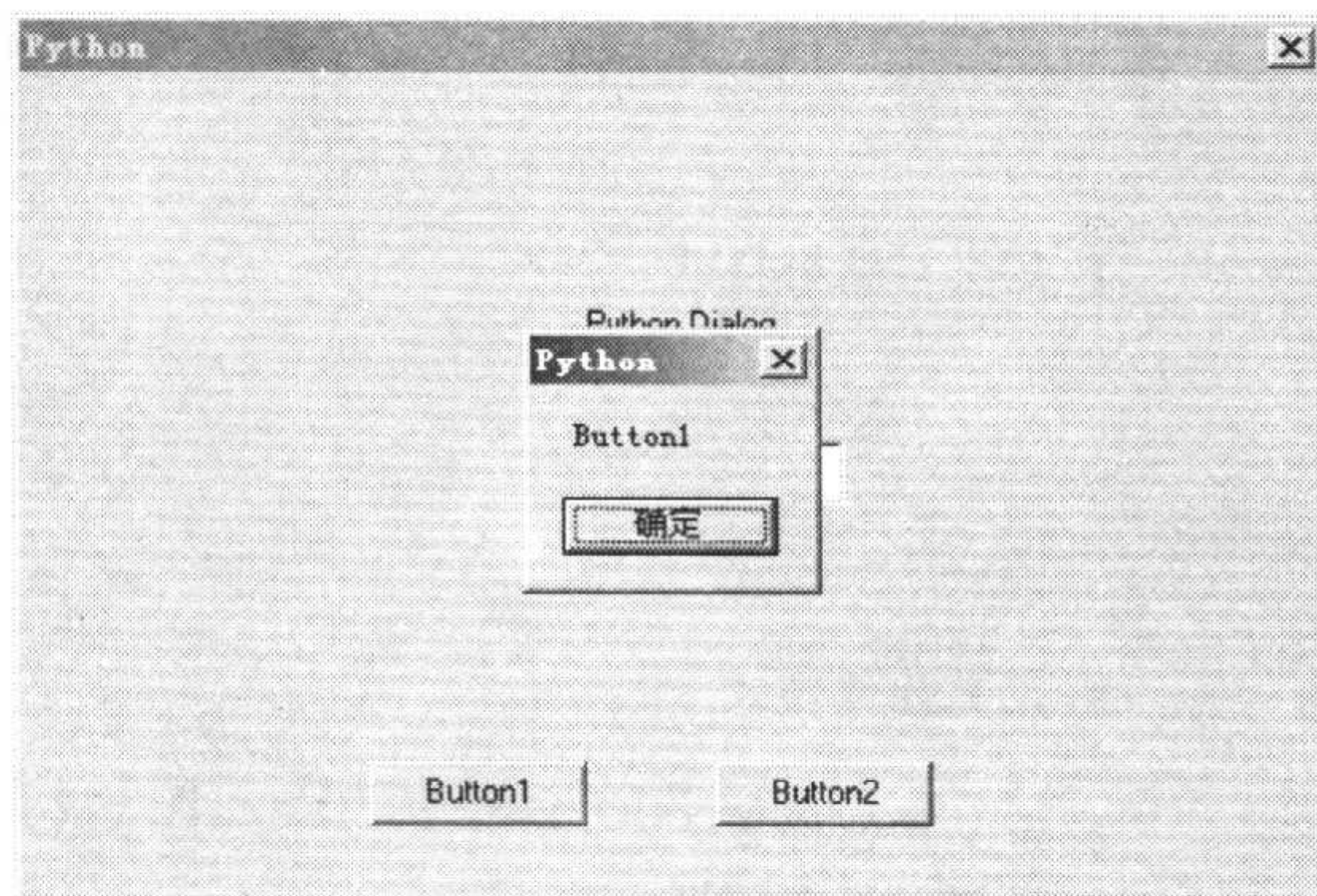


图 11-6 单击 Button1 按钮

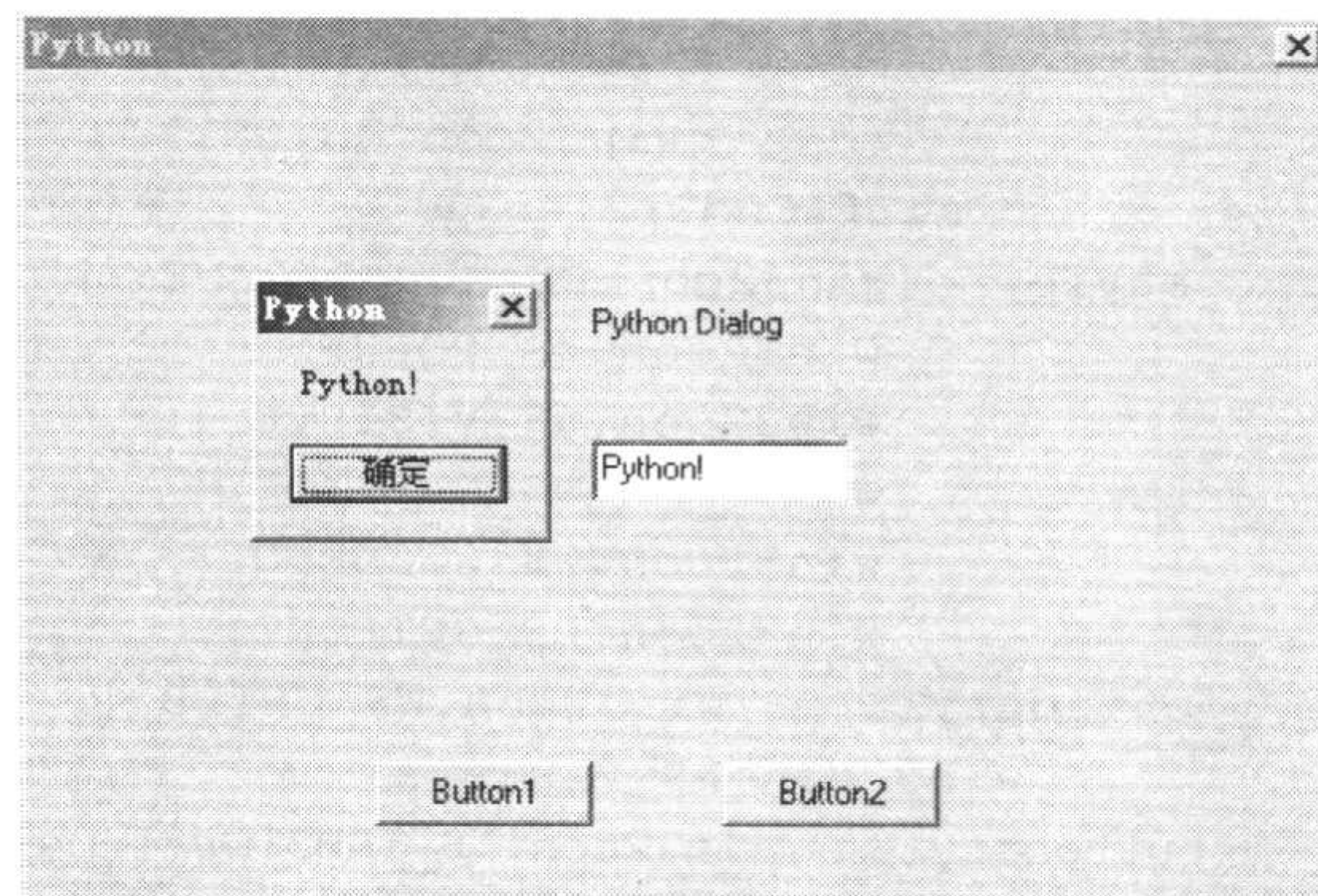


图 11-7 单击 Button2 按钮

11.3 菜单

菜单是 GUI 程序中重要的组成部分。通过菜单用户可以使用程序提供的功能，例如打开文件、关闭文件等。对于较为庞大的软件都由很多菜单项来完成各种功能。

11.3.1 创建菜单

通过使用 PythonWin 相关模块提供的函数，可以在脚本中动态地创建菜单。然后将创建的菜单添加到窗口中。

1. 创建普通菜单

使用 win32ui 模块中的 CreateMenu 可以动态地创建菜单，其返回值为 PyCMenu 对象。使用 PyCMenu 对象的 AppendMenu 方法可以向其中添加菜单。其原型如下所示。

```
AppendMenu(flags, id, value)
```

其参数含义如下。

- flags: 菜单样式。
- id: 菜单的 ID。
- value: 菜单名称。

其中参数 flags 应为 win32con.MF_POPUP、win32con.MF_SEPARATOR、win32con.MF_STRING、win32con.MF_UNCHECKED 等。value 可以为 None，但如果 value 为字符串，则 flags 中必须包含 win32con.MF_STRING 标志。

如下所示的 CreateMenu.py 脚本使用 CreateMenu 创建几种菜单。

```
# -*- coding:utf-8 -*-
# file: UseMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__(self):
        window.Wnd.__init__(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
            win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
            'MFC GUI', WS_OVERLAPPEDWINDOW, \
            (10, 10, 800, 500), None, 0, None)
        # 创建菜单对象
        submenu = win32ui.CreateMenu()
        menu = win32ui.CreateMenu()
        # 向菜单中添加 Open，其中&表示可以使用 Alt+&后的字母访问该菜单命令
```



```

submenu.AppendMenu(MF_STRING,1051,'&Open')
# 向菜单中添加 Close
submenu.AppendMenu(MF_STRING,1052,'&Close')
# 向菜单中添加 Save
submenu.AppendMenu(MF_STRING,1053,'&Save')
# 将上面的菜单添加到 File 菜单中
menu.AppendMenu(MF_STRING|MF_POPUP,submenu.GetHandle(), '&File')
submenu = win32ui.CreateMenu()
# 向菜单中添加 Copy
submenu.AppendMenu(MF_STRING,1054,'&Copy')
# 向菜单中添加 Paste
submenu.AppendMenu(MF_STRING,1055,'&Paste')
# 向菜单中添加分隔符
submenu.AppendMenu(MF_SEPARATOR,1056,None)
# 向菜单中添加 Cut
submenu.AppendMenu(MF_STRING,1057,'C&ut')
# 将上面的菜单添加到 Edit 菜单中
menu.AppendMenu(MF_STRING|MF_POPUP,submenu.GetHandle(), '&Edit')
submenu = win32ui.CreateMenu()
# 向菜单中添加 Tools
submenu.AppendMenu(MF_STRING,1058,'Tools')
# 向菜单中添加 Setting
submenu.AppendMenu(MF_STRING|MF_GRAYED,1059,'Setting')
m = win32ui.CreateMenu()
# 将上面的菜单添加到 Option 菜单中
m.AppendMenu(MF_STRING|MF_POPUP|MF_CHECKED,submenu.GetHandle(), 'Option')
# 将 Option 菜单添加到 Other 菜单中
menu.AppendMenu(MF_STRING|MF_POPUP,m.GetHandle(), '&Other')
# 将菜单添加到窗口中
self._obj_.SetMenu(menu)
# 重载 OnClose 方法
def OnClose(self):
    self.EndModalLoop(0)

w = MyWnd()
w.ShowWindow()
w.UpdateWindow()
w.RunModalLoop(1)

```

生成窗口对象
 # 显示窗口
 # 刷新窗口
 # 进入消息循环

运行脚本 CreateMenu.py 后将创建如图 11-8、图 11-9、图 11-10 所示的菜单。

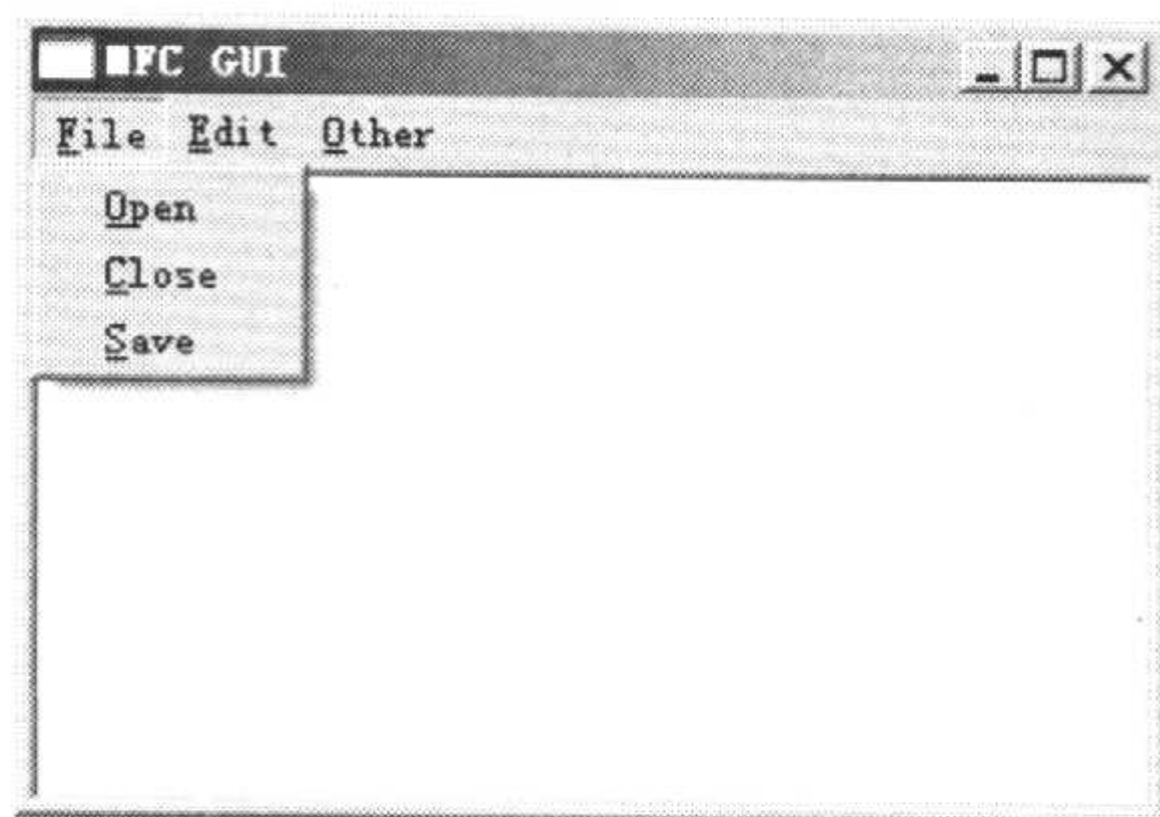


图 11-8 File 菜单

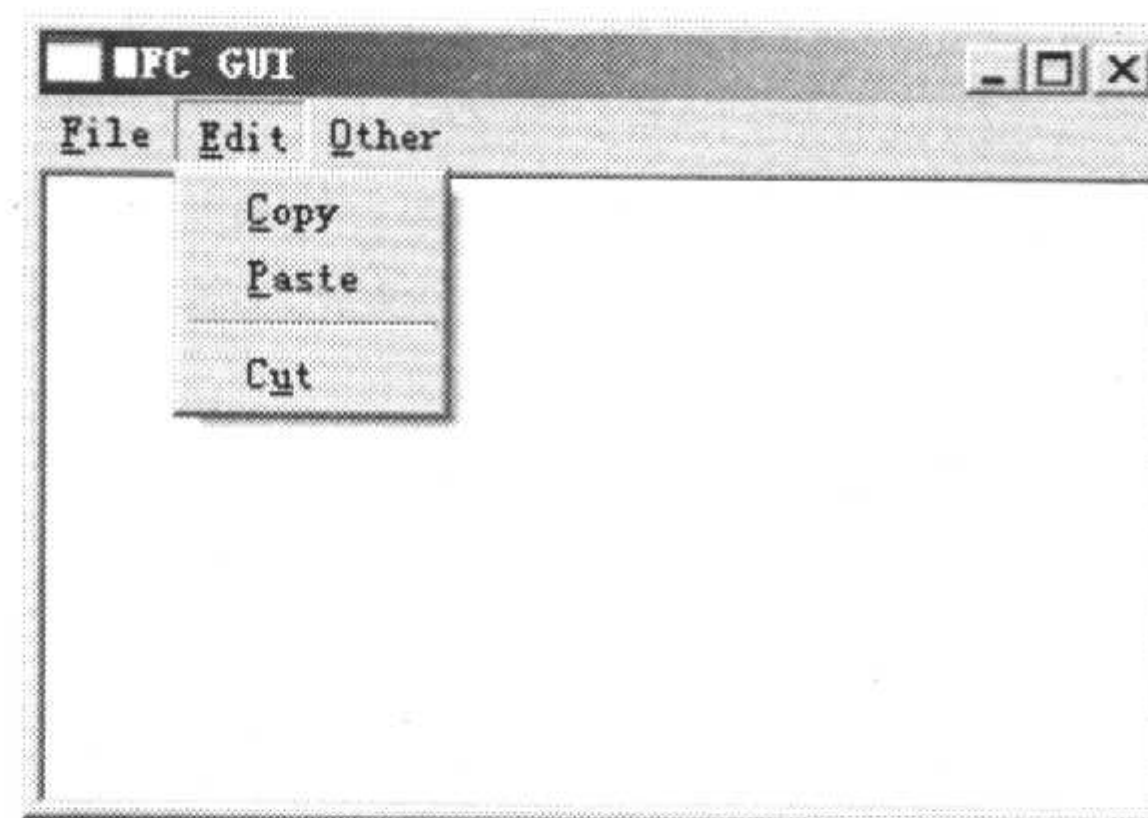


图 11-9 Edit 菜单



图 11-10 Other 菜单

2. 创建弹出式菜单

使用 win32ui 模块中的 CreatePopupMenu 可以动态地创建菜单，其返回值同 CreateMenu 一样为 PyCMenu 对象。在使用 CreatePopupMenu 创建完菜单后，就可以用 PyCMenu 对象的 AppendMenu 方法向其中添加菜单。如下所示的 PopupMenu.py 创建一个简单的窗口，当右击窗口时将弹出一个菜单。

```
# -*- coding:utf-8 -*-
# file: PopupMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__(self):
        window.Wnd.__init__(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
            win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
            'MFC GUI', WS_OVERLAPPEDWINDOW, \
            (10, 10, 800, 500), None, 0, None)
        # 捕获右键单击消息
        self.HookMessage(self.OnRClick, WM_RBUTTONDOWN)
    # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    # 处理单击鼠标右键消息
    def OnRClick(self, param):
        submenu = win32ui.CreatePopupMenu()
        submenu.AppendMenu(MF_STRING, 1054, 'Copy')           # 向菜单中添加 Copy
        submenu.AppendMenu(MF_STRING, 1055, 'Paste')         # 向菜单中添加 Paste
        submenu.AppendMenu(MF_SEPARATOR, 1056, None)         # 向菜单中添加分隔符
        submenu.AppendMenu(MF_STRING, 1057, 'Cut')           # 向菜单中添加 Cut
        flag = TPM_LEFTALIGN|TPM_LEFTBUTTON|TPM_RIGHTBUTTON # 弹出式菜单样式
        # param 为系统向 OnRClick 函数传递的参数，其为一个元组，其第 6 项为鼠标 x, y 坐标组成的元组
        submenu.TrackPopupMenu(param[5], flag, self)
w = MyWnd()                                                # 生成窗口对象
```



```
w.ShowWindow() # 显示窗口
w.UpdateWindow() # 刷新窗口
w.RunModalLoop(1) # 进入消息循环
```

运行脚本 PopupMenu.py 后，在创建的窗口中右击将弹出如图 11-11 所示的菜单。

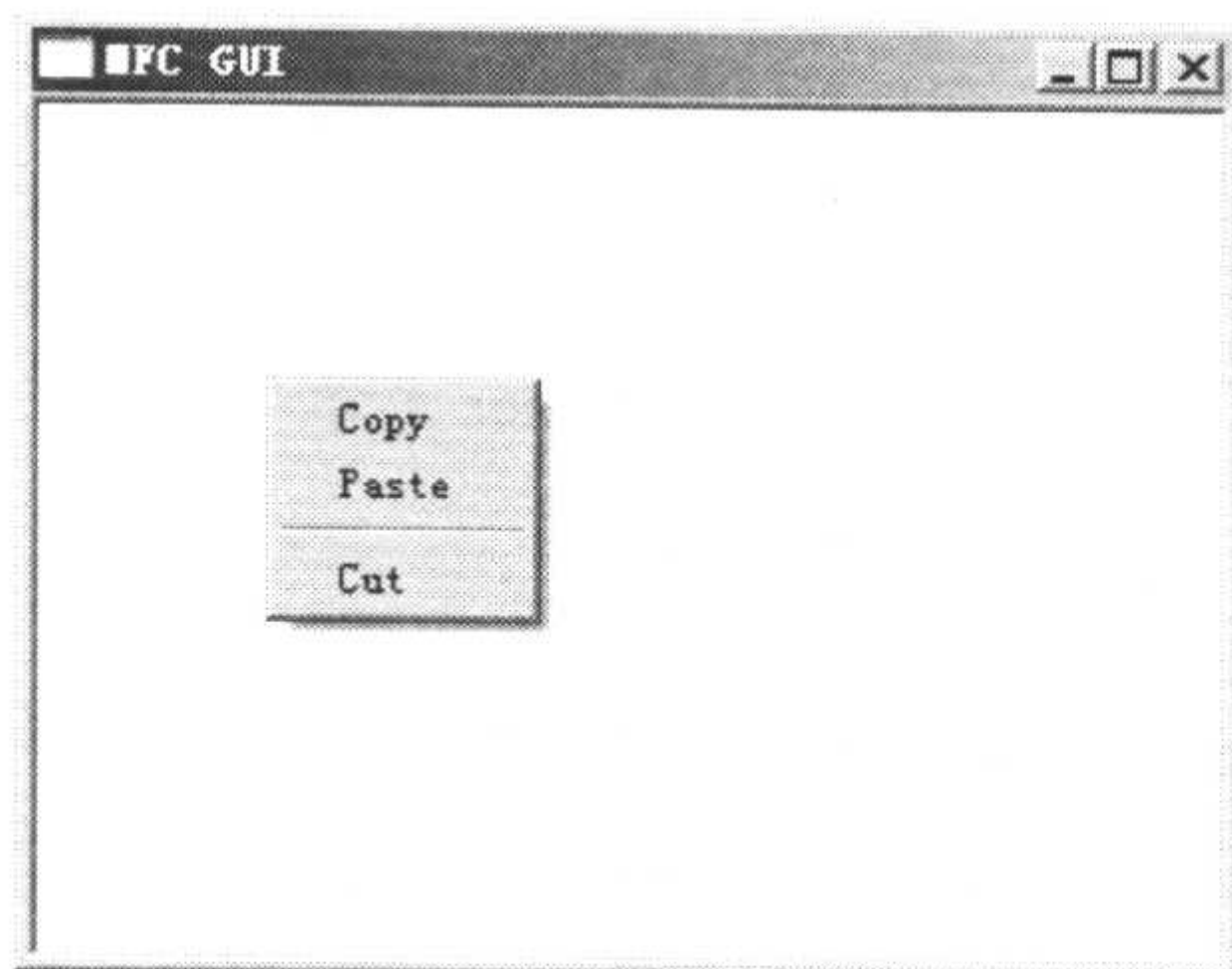


图 11-11 弹出式菜单

11.3.2 使用 DLL 中的菜单

11.3.1 节讲述了在 Python 脚本中直接创建菜单。如果窗口中菜单较多，那么创建菜单的代码将很多，而且创建过程又十分繁琐，同使用 DLL 文件中的对话框资源一样，也可以在 Python 中直接载入 DLL 文件中的菜单。使用 win32ui.LoadMenu 函数可以载入 DLL 文件中的菜单。其函数原型如下所示。

```
LoadMenu(id, dll)
```

其参数含义如下。

- idRes: 菜单的 ID。
- dll: 使用 win32ui.LoadLibrary 载入 DLL 文件的返回值。

可以直接使用上一节中创建的“Res”工程，向其中添加菜单编译完成后，打开工程中的“Resource.h”文件查看菜单的 ID。如下所示的 DllMenu.py 脚本载入 DLL 文件中的菜单，并将其添加到窗口中。

```
# -*- coding:utf-8 -*-
# file: DllMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__(self):
        window.Wnd.__init__(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
            win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
```


第11章 使用PythonWin编写GUI

```

        'MFC GUI', WS_OVERLAPPEDWINDOW, \
        (10, 10, 800, 500), None, 0, None)
    dll = win32ui.LoadLibrary('Res.dll')
    m = win32ui.LoadMenu(7001,dll)
    self._obj_.SetMenu(m)
# 重载 OnClose 方法
def OnClose(self):
    self.EndModalLoop(0)
# 重载 OnPaint 方法, 在窗口中输出 "MFC GUI"
def OnPaint(self):
    dc,ps = self.BeginPaint()
    dc.DrawText('MFC GUI',
        self.GetClientRect(),
        DT_SINGLELINE | DT_CENTER | DT_VCENTER)
    self.EndPaint(ps)

w = MyWnd()
w.ShowWindow()
w.UpdateWindow()
w.RunModalLoop(1)

```

载入 DLL 文件
载入菜单
添加菜单

生成窗口对象
显示窗口
刷新窗口
进入消息循环

运行脚本 DllMenu.py 后, 将创建如图 11-12~图 11-14 所示的菜单。

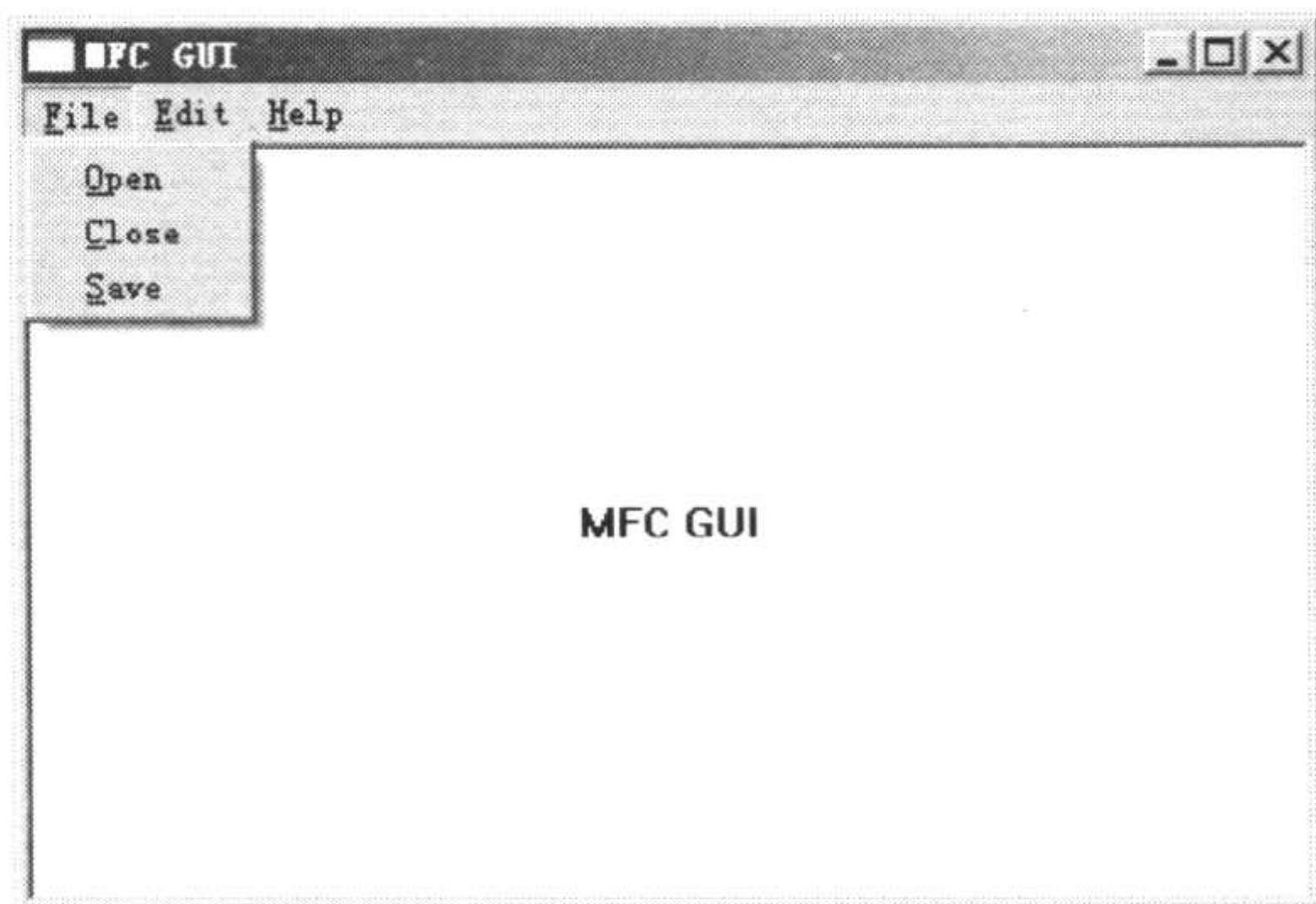


图 11-12 File 菜单

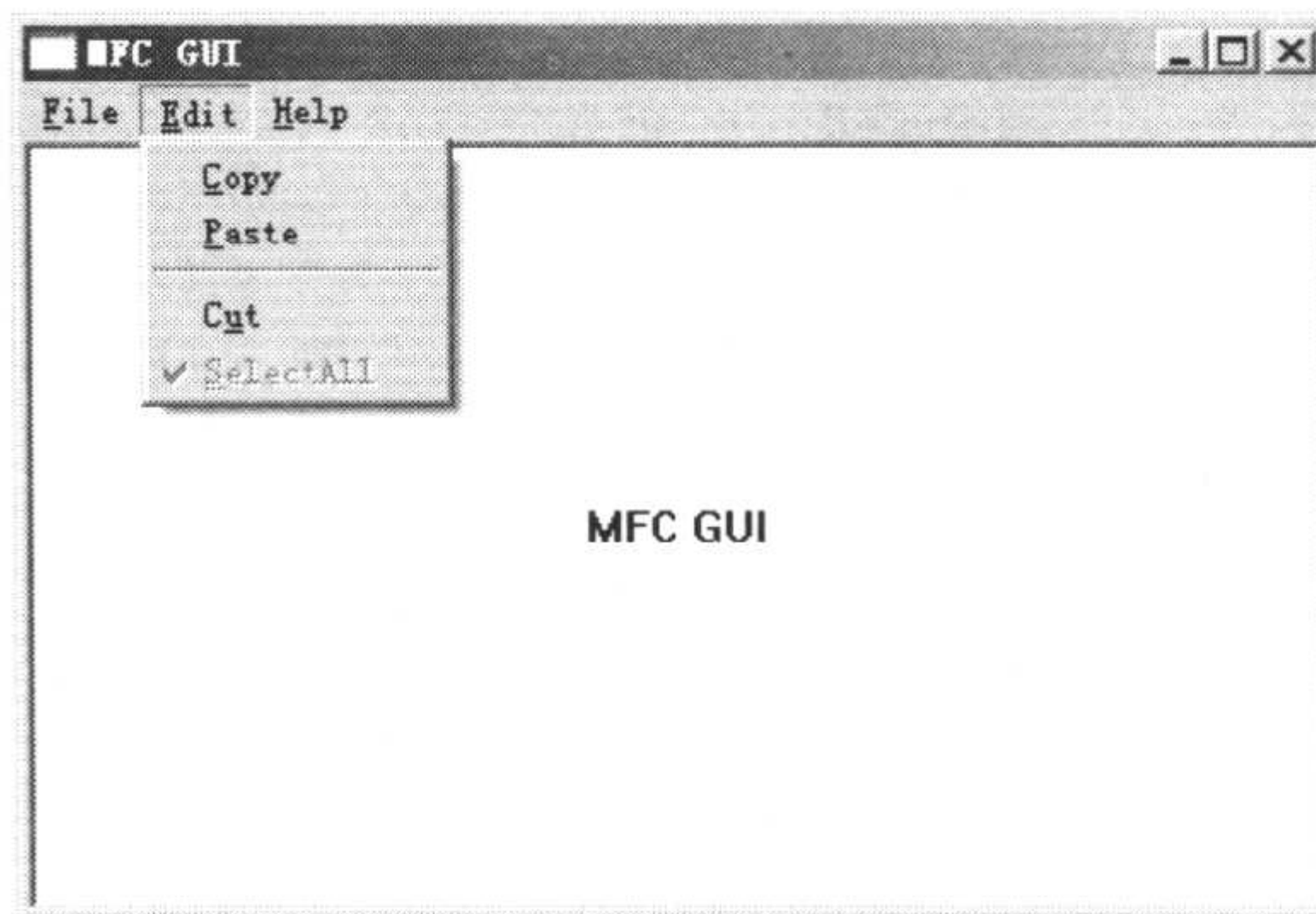


图 11-13 Edit 菜单

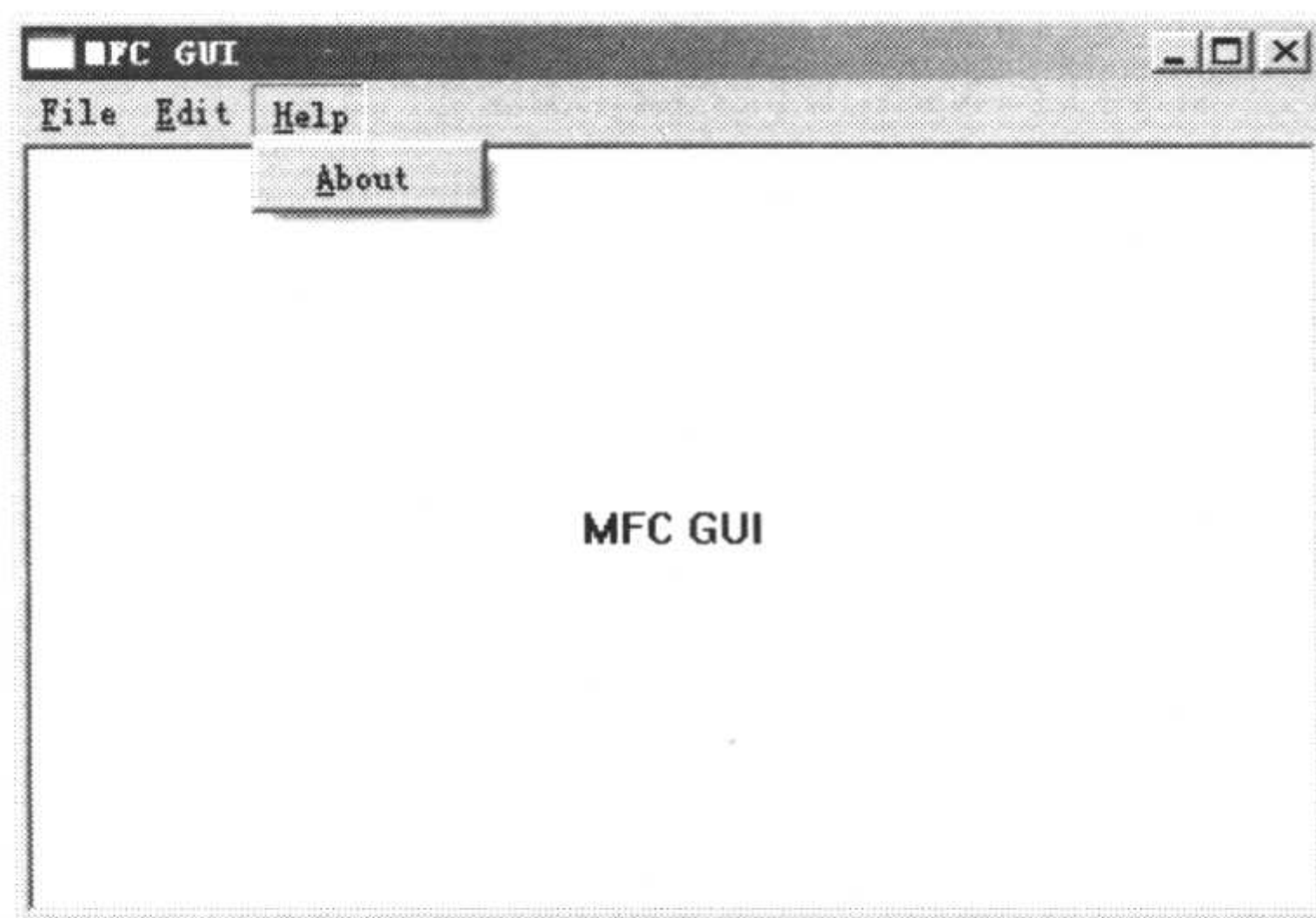


图 11-14 Help 菜单

11.3.3 处理菜单消息

菜单消息的处理与按钮消息的处理过程一样。只要使用 HookCommand 方法设置相应菜单的消息处理方法即可。如下所示的 MenuCmd.py 脚本使用 HookCommand 方法处理菜单的消息。

```
# -*- coding:utf-8 -*-
# file: UseMenu.py
#
import win32ui
import win32api
from win32con import *
from pywin.mfc import window
# 定义窗口类
class MyWnd(window.Wnd):
    def __init__(self):
        window.Wnd.__init__(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE, \
            win32ui.RegisterWndClass(0, 0, COLOR_WINDOW + 1), \
            'MFC GUI', WS_OVERLAPPEDWINDOW, \
            (10, 10, 800, 500), None, 0, None)
        # 创建菜单对象
        submenu = win32ui.CreateMenu()
        menu = win32ui.CreateMenu()
        # 向菜单中添加 Open, 其中&表示可以使用 Alt+&后的字母访问该菜单命令
        submenu.AppendMenu(MF_STRING, 1051, '&Open')
        # 向菜单中添加 Close
        submenu.AppendMenu(MF_STRING, 1052, '&Close')
        # 向菜单中添加 Save
        submenu.AppendMenu(MF_STRING, 1053, '&Save')
        # 将上面的菜单添加到 File 菜单中
        menu.AppendMenu(MF_STRING|MF_POPUP, submenu.GetHandle(), '&File')
        # 将菜单添加到窗口中
        self._obj_.SetMenu(menu)
        # 设置菜单处理消息
        self.HookCommand(self.MenuClick, 1051)
        self.HookCommand(self.MenuClick, 1052)
        self.HookCommand(self.MenuClick, 1053)
        # 重载 OnClose 方法
    def OnClose(self):
        self.EndModalLoop(0)
    def MenuClick(self, lParam, wParam):
        # 根据 lParam 参数判断单击的菜单
        if lParam == 1051:
            self.MessageBox('Open', 'Python', MB_OK)
        elif lParam == 1053:
            self.MessageBox('Save', 'Python', MB_OK)
        else:
```

第11章 使用PythonWin编写GUI

```
self.OnClose()

w = MyWnd()
w.ShowWindow()
w.UpdateWindow()
w.RunModalLoop(1)
```

生成窗口对象
显示窗口
刷新窗口
进入消息循环

运行 MenuCmd.py 脚本，单击菜单 **【File】 | 【Open】** 命令，如图 11-15 所示。单击菜单 **【File】 | 【Save】** 命令，如图 11-16 所示。单击菜单 **【File】 | 【Close】** 命令，将关闭窗口。

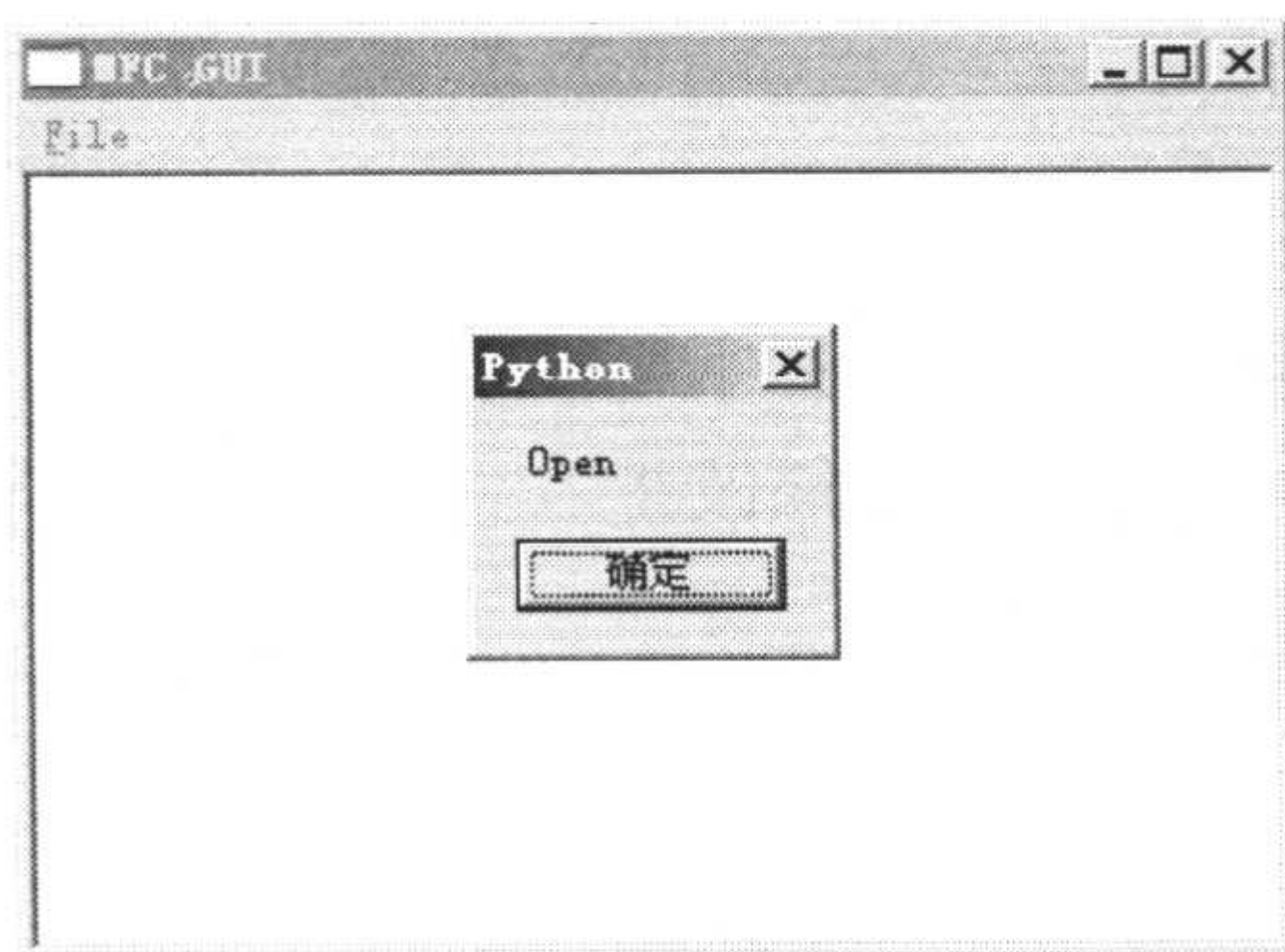


图 11-15 单击 **【Open】** 菜单命令

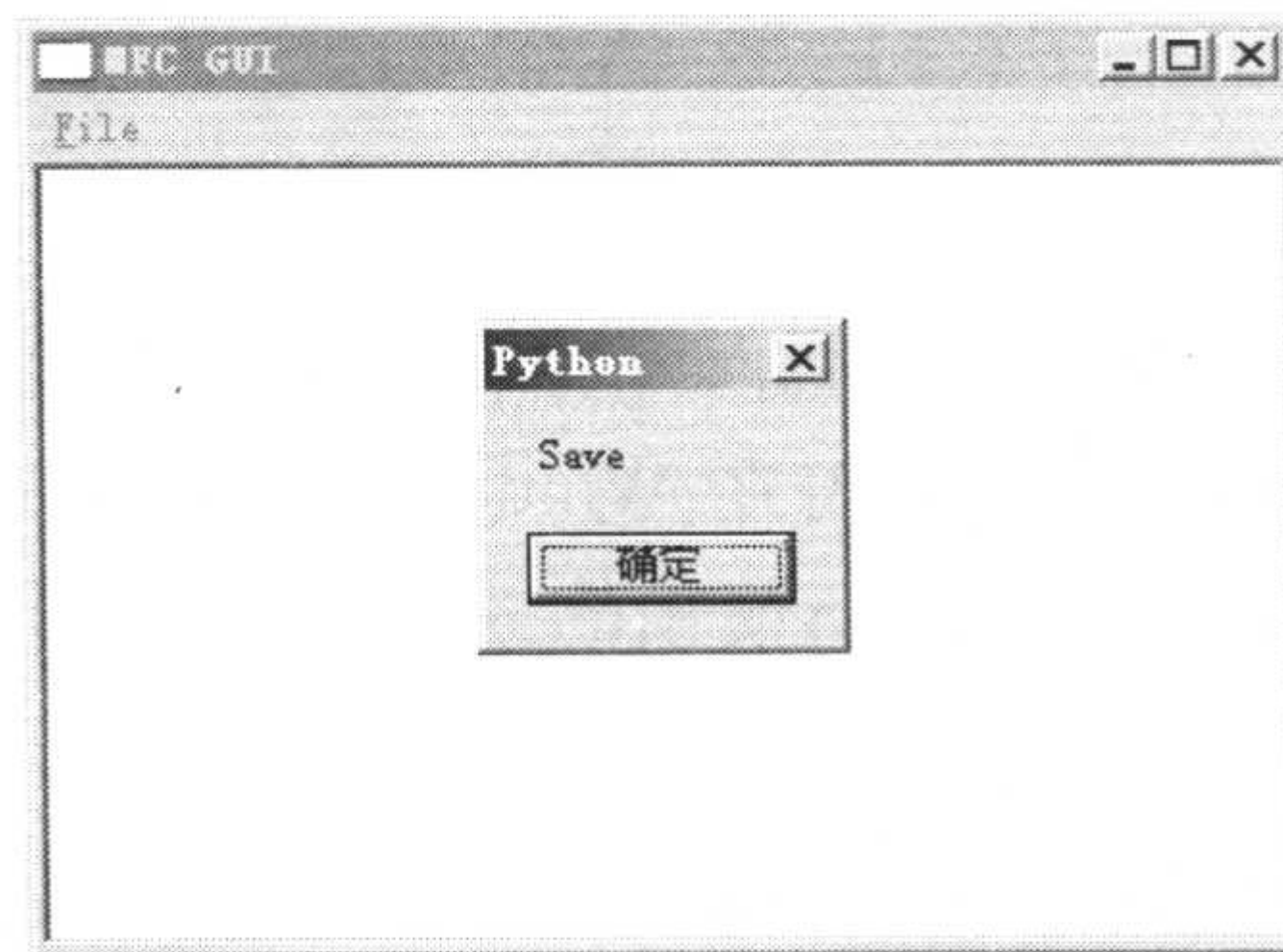


图 11-16 单击 **【Save】** 菜单命令

第 12 章 使用 Tkinter 编写 GUI

Tkinter 是 Python 自带的用于 GUI 编程的模块。Tkinter 是对图形库 TK 的封装。Tkinter 是跨平台的，这意味着在 Windows 下编写的脚本，可以不加修改地在 Linux、UNIX 等系统下运行。而上一章中所使用的 MFC 仅适用于 Windows 平台。

12.1 Tkinter 概述

Tkinter 的优势在于其可移植性，使用它可以创建完整的 GUI 程序。在 Tkinter 中，文本框、按钮、标签等在上一章中所提到的都被称之为组件 (widget)。Tkinter 中的组件相当于上一章中所提到的控件。Tkinter 是 Python 的一个模块，可以像其他模块一样在 Python 的交互式 shell 中，或者“.py”脚本中被导入。Tkinter 模块被导入后即可使用 Tkinter 模块中的函数、方法等。

12.1.1 创建简单的窗口

使用 Tkinter 创建图形界面时要首先导入 Tkinter 模块。可以在 Python 的交互式 shell 中，输入如下所示的语句验证 Python 是否安装了 Tkinter 模块。

```
import Tkinter
```

如果上述语句执行成功，则表示已经安装了 Tkinter 模块。在编写脚本时只要在使用 import 语句导入 Tkinter 模块时即可使用 Tkinter 模块中的函数、对象等进行 GUI 编程了。

在使用 Tkinter 模块时，首先要使用 Tkinter.Tk 生成一个主窗口对象，然后才能使用 Tkinter 模块中其他的函数、方法等。当生成主窗口以后可以向其添加组件，或者直接调用其 mainloop 方法进行消息循环。如下所示的 TkinterWindow.py 脚本仅创建了一个简单的窗口而没有使用组件。

```
# -*- coding:utf-8 -*-
# file: TkinterWindow.py
#
import Tkinter                                     # 导入 Tkinter 模块
root = Tkinter.Tk()                                # 生成 root 主窗口
root.mainloop()                                    # 进入消息循环
```

运行脚本 TkinterWindow.py 后，将创建如图 12-1 所示的窗口。

上述脚本同上一章中使用 MFC 创建窗口的脚本相比更为简单，仅使用 3 条语句就创建了一个简单的 GUI 窗口。当完成窗口机器内部的组件的创建工作时，也要进入消息循环中，以处理窗口及其内部组件的事件。

如果需要在 Tkinter 的窗口、组件中显示中文，除了在“.py”脚本文件中的首行添加“# -*- coding:utf-8 -*-”指明字符编码以外，还应该将脚本保存成“UTF-8”的编码格式。如果使用记事本，则可以单击【文件】|【另存为】命令，选择另存为对话框中下方【编码】下拉列表框中的“UTF-8”选项。然后单击【保存】按钮，即可在 Tkinter 的窗口或组件中使用中文。

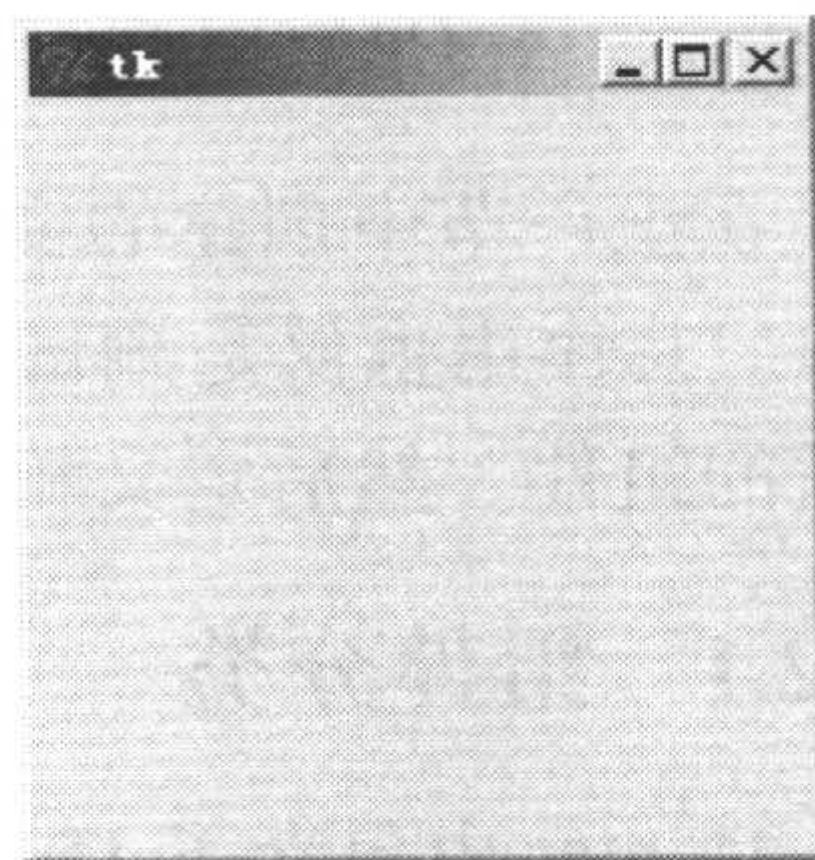


图 12-1 简单的 Tkinter 窗口

12.1.2 向窗口中添加组件

在 Tkinter 中组件也是像主窗口一样使用 Tkinter 模块中相应的组件函数生成的。组件生成后就可以使用 pack 方法、grid 方法，或者 place 方法将其添加到窗口中。如下所示的 HelloTkinter.py 将上一小节中的例子改写，向窗口中添加了标签和按钮组件。

<pre># -*- coding:utf-8 -*- # file: HelloTkinter.py # import Tkinter root = Tkinter.Tk() label= Tkinter.Label(root, text="Hello, Tkinter!") label.pack() button1 = Tkinter.Button(root, text="Button1") button1.pack(side=Tkinter.LEFT) button2 = Tkinter.Button(root, text="Button2") button2.pack(side=Tkinter.RIGHT) root.mainloop()</pre>	<pre># 导入 Tkinter 模块 # 生成 root 主窗口 # 生成标签 # 将标签添加到 root 主窗口 # 生成 button1 # 将 button1 添加到 root 主窗口 # 生成 button2 # 将 button2 添加到 root 主窗口 # 进入消息循环</pre>
---	--

运行 HelloTkinter.py 脚本后将创建如图 12-2 所示的窗口。



图 12-2 向窗口中添加组件

在使用 Tkinter 向窗口中添加组件时，如果不指定组件的位置信息，Tkinter 将自动将组件安排在合适的位置。组件的位置也可以精确地控制，达到所需要的效果。在上述的脚本中，使用了标签和按钮组件。除此以外，Tkinter 还提供了其他的组件，用以适应不同的需要。

运行 HelloTkinter.py 后，单击两个按钮均无反应，这是因为在脚本中未添加按钮组件单击事件的处理函数。关于组件的事件处理将在后边的章节进行讲解，此处仅给出一个简单的例子。

12.2 使用组件

在上一节中创建的窗口实际上是存放组件的一个“容器”。如果仅创建一个不包含组件的窗口，其作用也仅是测试 Tkinter 模块。当窗口创建好以后，应根据脚本的功能向窗口中添加合适的组件。然后定义相关实际的处理函数，这样才算一个完整的 GUI 程序。

12.2.1 组件分类

Tkinter 中包含了 15 种核心组件，用以实现不同的功能。详细的组件如表 12-1 所示。

表 12-1 Tkinter 中的组件

组 件 名 称	描 述
Button	按钮
Canvas	绘图形组件，可以在其中绘制图形
Checkbutton	复选框
Entry	文本框（单行）
Frame	框架，将几个组件组成一组
Label	标签，可以显示文字或者图片
Listbox	列表框
Menu	菜单
Menubutton	Menubutton 的功能完全可以使用 Menu 替代
Message	与 Label 组件类似，但是可以根据自身大小将文本换行
Radiobutton	单选框
Scale	滑块
Scrollbar	滚动条
Text	文本框（多行）
Toplevel	用来创建子窗口容器组件

12.2.2 组件布局

在前边的例子中仅使用组件的 pack 方法将组件添加到窗口中，而未设置组件的位置。组件位置都是由 Tkinter 模块自动确定的。对于包含较多组件的窗口，为了组件布局合理可以通过向 pack 传递参数来设置组件在窗口中的位置。除了组件的 pack 方法以外，还可以使用 grid 方法和 place 方法来放置组件。组件的 pack 方法可以使用以下几个参数设置组件的位置属性。

- after：将组件置于其他组件之后。
- anchor：组件的对齐方式，顶对齐“n”，底对齐“s”，左对齐“w”，右对齐“e”。

- before: 将组件置于其他组件之前。
- side: 组件在主窗口的位置, 可以为“top”、“bottom”、“left”、“right”。

组件的 grid 方法使用行列的方法放置组件的位置。该方法可以使用以下几个参数设置组件的位置属性。

- column: 组件所在的列起始位置。
- columnspan: 组件的列宽。
- row: 组件所在的行起始位置。
- rowspan: 组件的行宽。

组件的 place 方法相对较灵活, 可以直接使用坐标来放置组件的位置。该方法可以使用以下几个参数设置组件的位置属性。

- anchor: 组件对齐方式。
- x: 组件左上角的 x 坐标。
- y: 组件左上角的 y 坐标。
- relx: 组件相对于窗口的 x 坐标, 应为 $0 \sim 1$ 之间的小数。
- rely: 组件相对于窗口的 y 坐标, 应为 $0 \sim 1$ 之间的小数。
- width: 组件的宽度。
- height: 组件的高度。
- relwidth: 组件相对于窗口的宽度, 应为 $0 \sim 1$ 之间的小数。
- relheight: 组件相对于窗口的高度, 应为 $0 \sim 1$ 之间的小数。

12.2.3 使用按钮

使用 Tkinter.Button 时向其传递参数可以控制按钮的属性, 例如按钮上文本的颜色、按钮的颜色、按钮的大小以及按钮的状态。常用的控制参数有如下几个。

- anchor: 指定按钮上文本的位置。
- background (bg): 指定按钮的背景色。
- bitmap: 指定按钮上显示的位图。
- borderwidth (bd): 指定按钮边框的宽度。
- command: 指定按钮消息的回调函数。
- cursor: 指定鼠标移动到按钮上的指针样式。
- font: 指定按钮上文本的字体。
- foreground (fg): 指定按钮的前景色。
- height: 指定按钮的高度。

- image: 指定按钮上显示的图片。
- state: 指定按钮的状态。
- text: 指定按钮上显示的文本。
- width: 指定按钮的宽度。

如下所示的 TkinterButton.py 脚本创建了各种不同的按钮。

```
# -*- coding:utf-8 -*-
# file: TkinterButton.py
#
import Tkinter                                     # 导入 Tkinter 模块
root = Tkinter.Tk()
button1 = Tkinter.Button(root,                    # 指定文本对齐方式
                           anchor = Tkinter.E,    # 指定按钮上的文本
                           text = 'Button1',       # 指定按钮的宽度, 相当于 40 个字符
                           width = 40,            # 指定按钮的高度, 相当于 5 行字符
                           height = 5)            # 将按钮添加到窗口
button1.pack()
button2 = Tkinter.Button(root,
                           text = 'Button2',
                           bg = 'blue')           # 指定按钮的背景色
button2.pack()
button3 = Tkinter.Button(root,
                           text = 'Button3',
                           width = 14,            # 指定按钮的宽度
                           height = 1)            # 指定按钮的高度
button3.pack()
button4 = Tkinter.Button(root,
                           text = 'Button4',
                           width = 60,
                           height = 5,
                           state = Tkinter.DISABLED) # 指定按钮为禁用状态
button4.pack()
root.mainloop()                                   # 进入消息循环
```

运行 TkinterButton.py 脚本后将创建如图 12-3 所示的 4 个按钮。

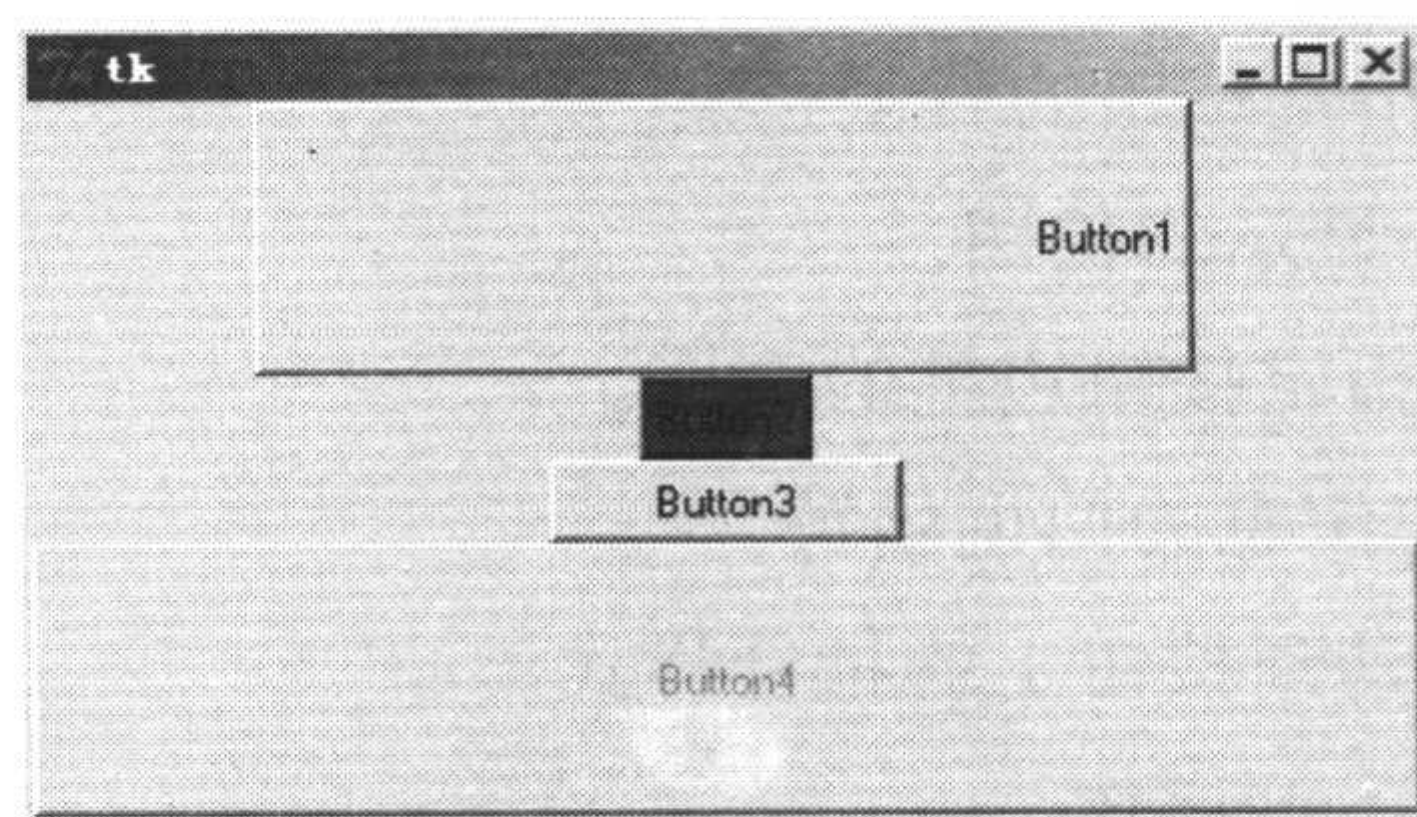


图 12-3 不同类型的按钮

12.2.4 使用文本框

文本框主要用来接收用户输入。使用 Tkinter.Entry 和 Tkinter.Text 可以创建单行文本框和多行文本框组件。通过向其传递参数可以设置文本框的背景色、大小、状态等。以下是其共有的几个控制参数。

- background (bg): 指定文本框的背景色。
- borderwidth (bd): 指定文本框边框的宽度。
- font: 指定文本框中文字的字体。
- foreground (fg): 指定文本框的前景色。
- selectbackground: 指定选定文本的背景色。
- selectforeground: 指定选定文本的前景色。
- show: 指定文本框中显示的字符, 若为 “*”, 表示文本框为密码框。
- state: 指定文本框的状态。
- width: 指定文本框的宽度。

如下所示的 TkinterEntry.py 创建了各种不同类型的文本框。

```
# -*- coding:utf-8 -*-
# file: TkinterEntry.py
#
import Tkinter                                # 导入 Tkinter 模块
root = Tkinter.Tk()
entry1 = Tkinter.Entry(root,                  # 生成单行文本框组件
                        show = '*',           # 输入文本框中的字符被显示为 “*”
                        )                     # 将文本框添加到窗口中
entry1.pack()
entry2 = Tkinter.Entry(root,                  # 输入文本框中的字符被显示为 “#”
                        show = '#',          # 将文本框的宽度设置为 50
                        width = 50)
entry2.pack()
entry3 = Tkinter.Entry(root,                  # 将文本框的背景色设置为红色
                        bg = 'red',           # 将文本框的前景色设置为蓝色
                        fg = 'blue')
entry3.pack()
entry4 = Tkinter.Entry(root,                  # 将选中文本的背景色设置为红色
                        selectbackground = 'red',
                        selectforeground = 'gray')
entry4.pack()
entry5 = Tkinter.Entry(root,                  # 将文本框设置为禁用
                        state = Tkinter.DISABLED)
entry5.pack()
edit1 = Tkinter.Text(root,                   # 生成多行文本框组件
                      selectbackground = 'red',
                      selectforeground = 'gray')
```



```
editl.pack()
root.mainloop() # 进入消息循环
```

运行 TkinterEntry.py 脚本后将创建如图 12-4 所示的 6 个文本框。

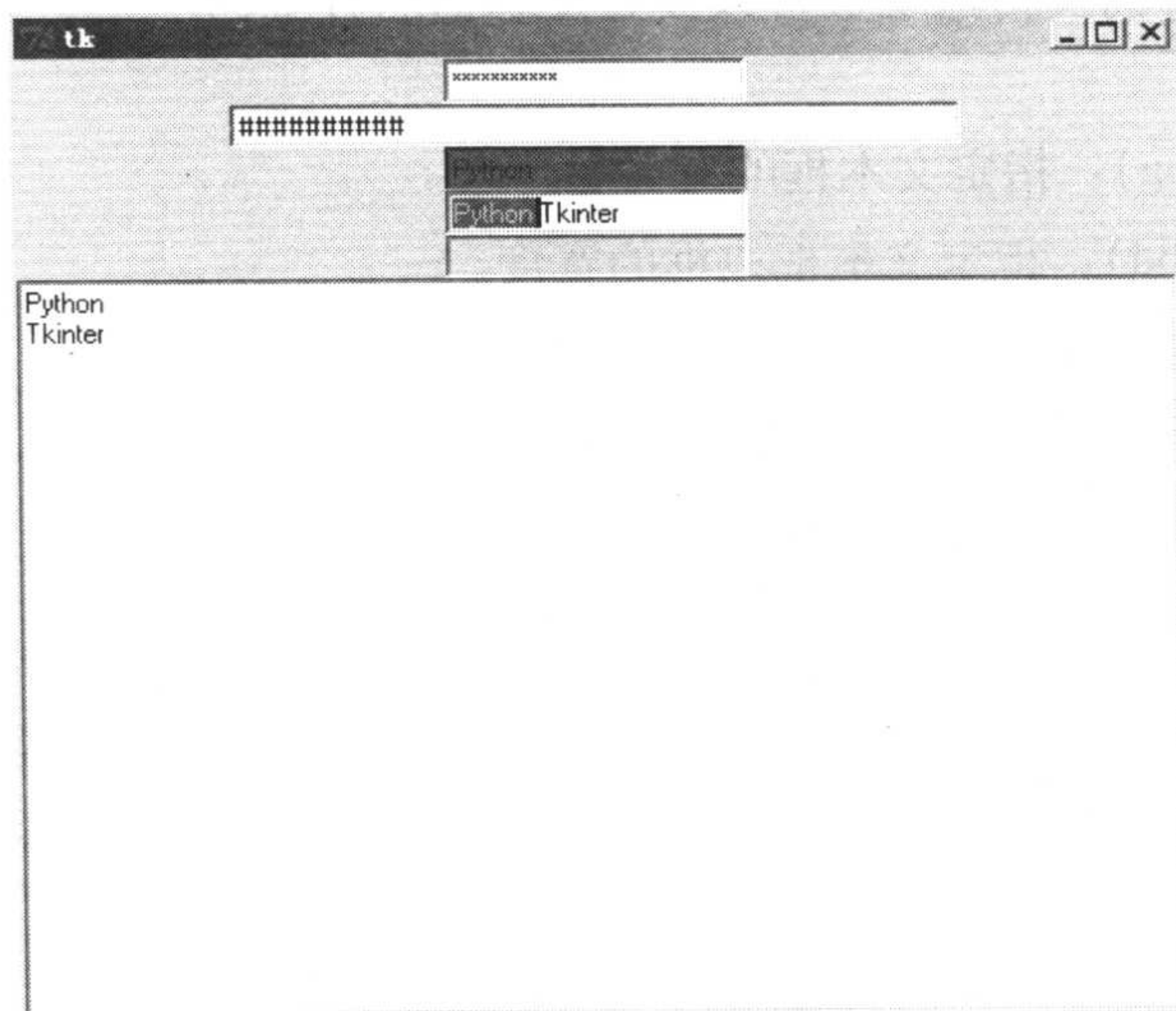


图 12-4 不同类型的文本框

12.2.5 使用标签

标签是提供在窗口中显示文本的组件。除了显示文本以外，标签还可以显示图片。使用 Tkinter.Label 可以创建标签组件。以下是其几种属性的控制参数。

- anchor: 指定标签中文本的位置。
- background (bg): 指定标签的背景色。
- borderwidth (bd): 指定标签的边框宽度。
- bitmap: 指定标签中的位图。
- font: 指定标签中文本的字体。
- foreground (fg): 指定标签的前景色。
- height: 指定标签的高度。
- image: 指定标签中的图片。
- justify: 指定标签中多行文本的对齐方式。
- text: 指定标签中的文本，可以使用“\n”表示换行。
- width: 指定标签的宽度。

如下所示的 TkinterLabel.py 创建了几种不同类型的标签组件。

第12章 使用Tkinter编写GUI

```

# -*- coding:utf-8 -*-
# file: TkinterLabel.py
#
import Tkinter                                # 导入Tkinter 模块
root = Tkinter.Tk()
label1 = Tkinter.Label(root,
                        anchor = Tkinter.E,      # 设置文本的位置
                        bg = 'blue',            # 设置标签背景色
                        fg = 'red',             # 设置标签前景色
                        text = 'Python',         # 设置标签中的文本
                        width = 30,              # 设置标签的宽度为 30
                        height = 5)             # 设置标签的高度为 5
label1.pack()
label2 = Tkinter.Label(root,
                        text = 'Python GUI\nTkinter', # 设置标签中的文本, 在字符串中使用换行符
                        justify = Tkinter.LEFT,      # 设置多行文本为左对齐
                        width = 30,
                        height = 5)
label2.pack()
label3 = Tkinter.Label(root,
                        text = 'Python GUI\nTkinter',
                        justify = Tkinter.RIGHT,      # 设置多行文本为右对齐
                        width = 30,
                        height = 5)
label3.pack()
label4 = Tkinter.Label(root,
                        text = 'Python GUI\nTkinter',
                        justify = Tkinter.CENTER,     # 设置多行文本为居中对齐
                        width = 30,
                        height = 5)
label4.pack()
root.mainloop()

```

运行 TkinterLabel.py 脚本后将创建如图 12-5 所示的 4 个不同类型的标签组件。

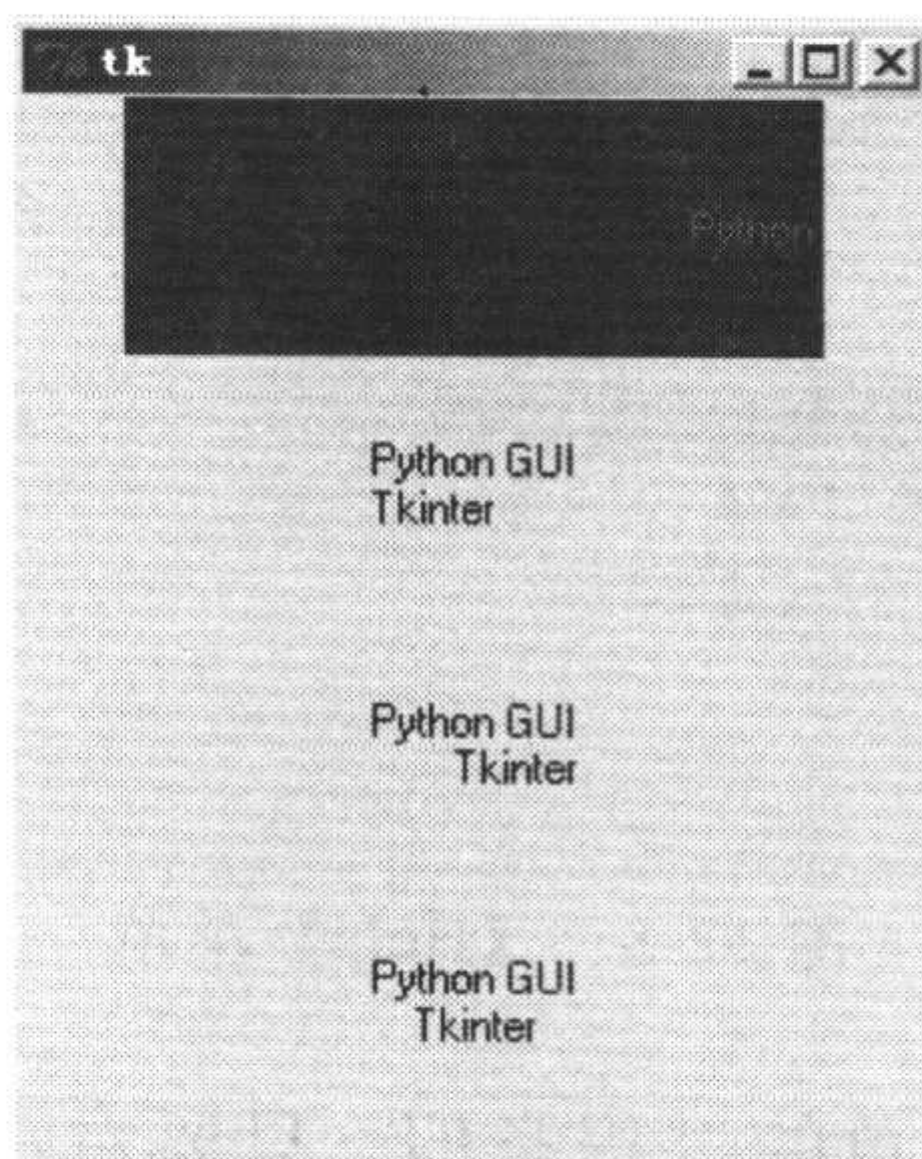


图 12-5 不同类型的标签

12.2.6 使用菜单

在 Tkinter 中菜单组件的添加与其他组件有所不同。菜单要使用所创建的主窗口的 config 方法添加到窗口中。如下所示的 TkinterMenu.py 向主窗口中添加了菜单。

```
# -*- coding:utf-8 -*-
# file: TkinterMenu.py
#
import Tkinter
root = Tkinter.Tk()
menu = Tkinter.Menu(root)
submenu = Tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="Open")
submenu.add_command(label="Save")
submenu.add_command(label="Close")
menu.add_cascade(label="File", menu=submenu)
submenu = Tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="Copy")
submenu.add_command(label="Paste")
submenu.add_separator()
submenu.add_command(label="Cut")
menu.add_cascade(label="Edit", menu=submenu)
submenu = Tkinter.Menu(menu, tearoff=0)
submenu.add_command(label="About")
menu.add_cascade(label="Help", menu=submenu)
root.config(menu=menu)
root.mainloop()
```

导入 Tkinter 模块

生成菜单

生成下拉菜单

向下拉菜单中添加 Open 命令

向下拉菜单中添加 Save 命令

向下拉菜单中添加 Close 命令

将下拉菜单添加到菜单中

生成下拉菜单

向下拉菜单中添加 Copy 命令

向下拉菜单中添加 Paste 命令

向下拉菜单中添加分隔符

向下拉菜单中添加 Cut 命令

将下拉菜单添加到菜单中

生成下拉菜单

向下拉菜单中添加 About 命令

将下拉菜单添加到菜单中

运行脚本后将创建如图 12-6~图 12-8 所示的菜单。

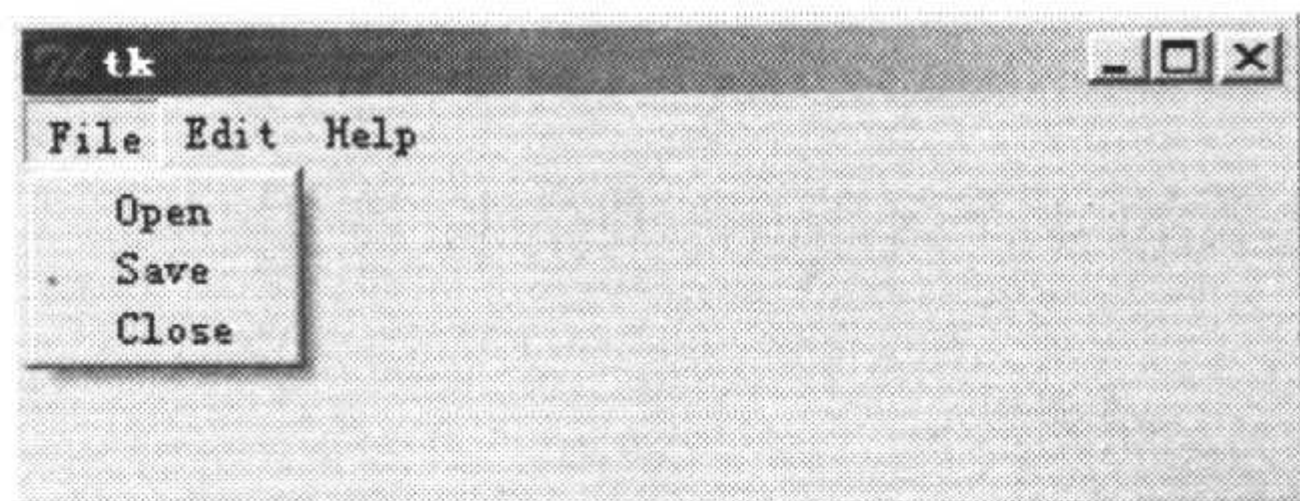


图 12-6 【File】菜单

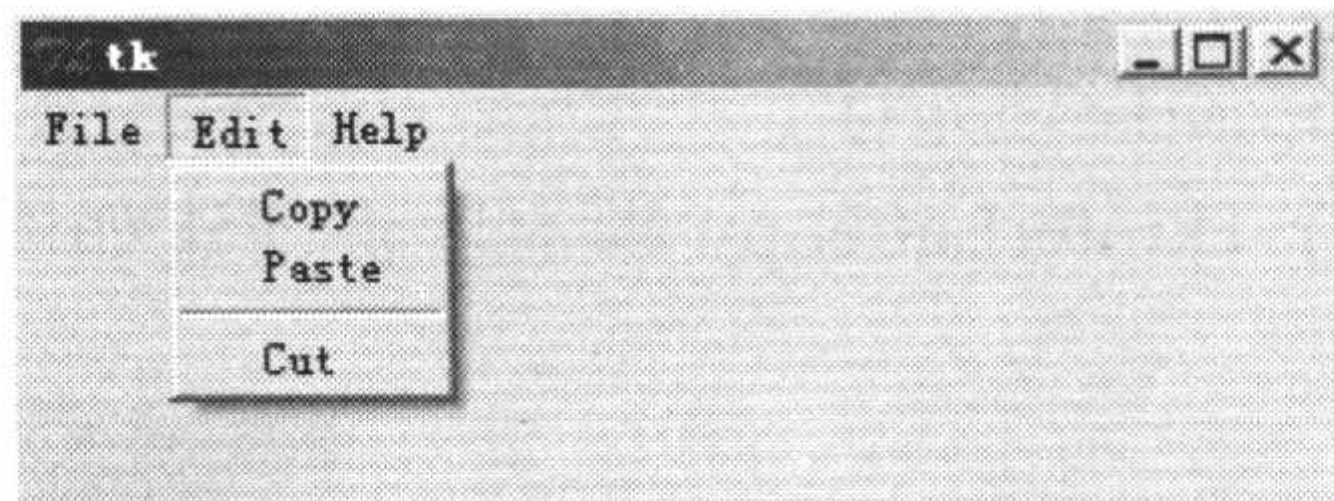


图 12-7 【Edit】菜单

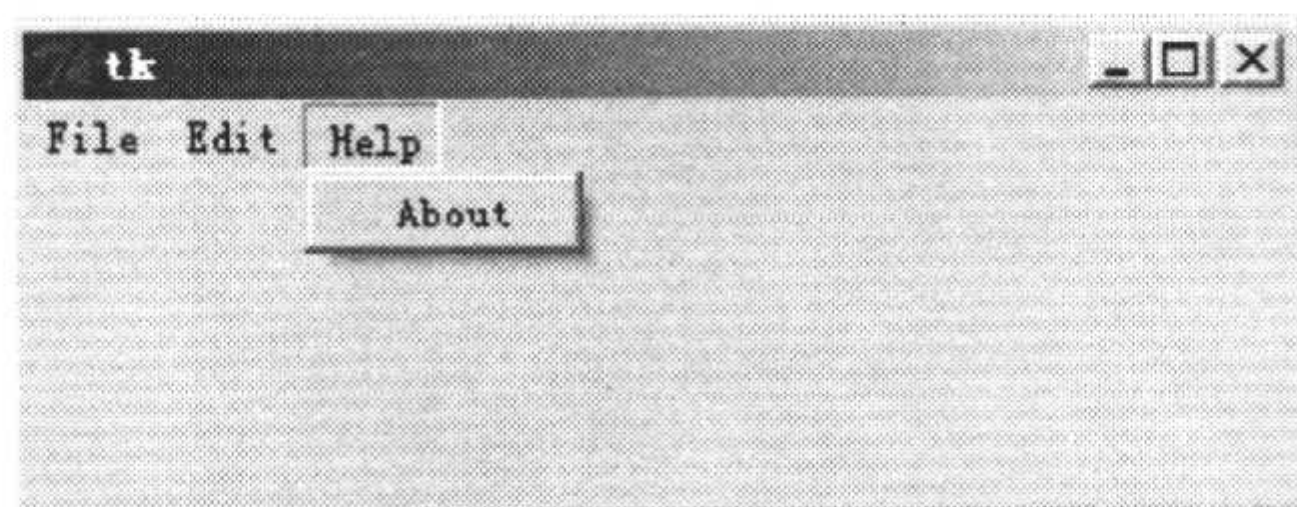


图 12-8 【Help】菜单

创建弹出式菜单和上述创建菜单的过程类似。只要右击后使用菜单的 post 方法显示菜单即可。如下所示的 TkinterPopupmenu.py 脚本创建一个简单的右击弹出式菜单。


```

# -*- coding:utf-8 -*-
# file: TkinterPopupmenu.py
#
import Tkinter
root = Tkinter.Tk()
menu = Tkinter.Menu(root, tearoff=0)
menu.add_command(label="Copy")
menu.add_command(label="Paste")
menu.add_separator()
menu.add_command(label="Cut")
def popupmenu(event):
    menu.post(event.x_root, event.y_root)
root.bind("<Button-3>", popupmenu)
root.mainloop()

```

创建菜单
 # 向弹出式菜单中添加 Copy 命令
 # 向弹出式菜单中添加 Paste 命令
 # 向弹出式菜单中添加分隔符
 # 向弹出式菜单中添加 Cut 命令
 # 定义右键事件处理函数
 # 显示菜单
 # 在主窗口中绑定右键事件

运行 TkinterPopupmenu.py 脚本后将创建如图 12-9 所示的弹出式菜单。

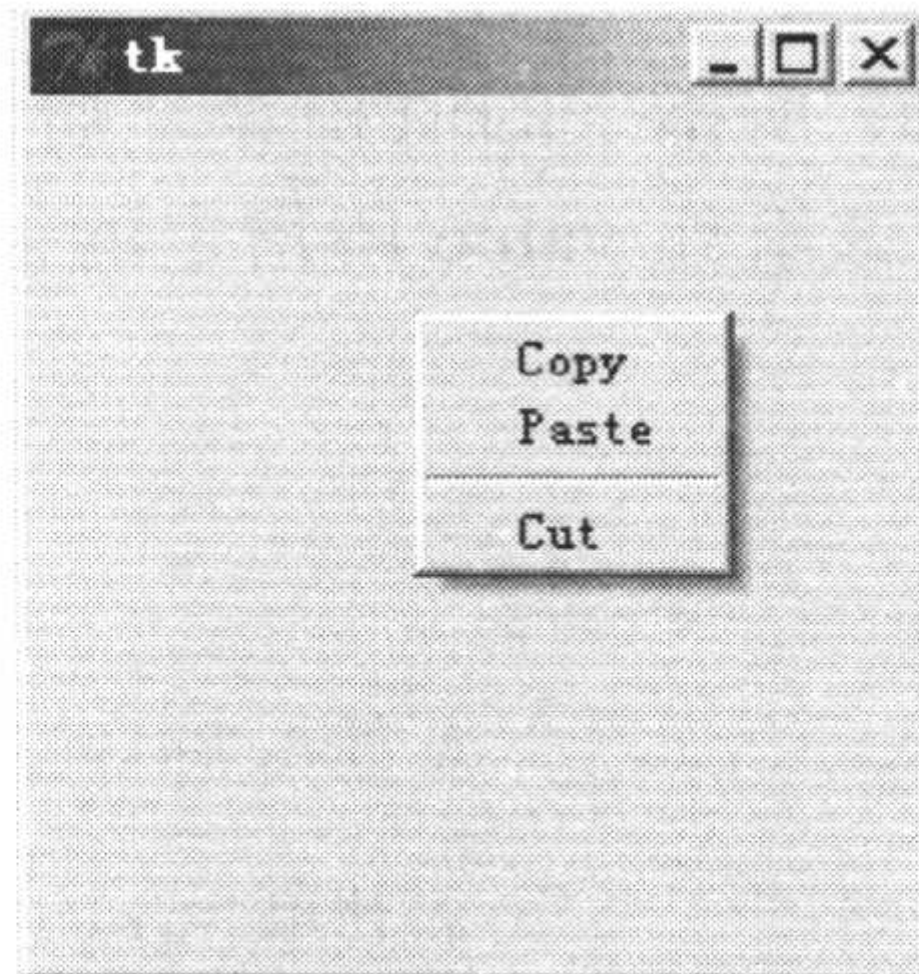


图 12-9 创建弹出式菜单

12.2.7 使用单选框和复选框

单选框往往用于一组互斥的选项。一组单选框中只有一个可以被选中。而复选框则由一个复选框组件来表示两种不同的状态，即被选中表示一种状态，未被选中表示另一种状态。使用 Tkinter.Radiobutton 和 Tkinter.Checkbutton 可以分别创建单选框和复选框。通过向其传递参数可以设置单选框和复选框的背景色、大小、状态等。以下是其共有的几个控制参数。

- anchor: 指定文本位置。
- background (bg): 指定背景色。
- borderwidth (bd): 指定边框的宽度。
- bitmap: 指定组件中的位图。
- font: 指定组件中文本的字体。
- foreground (fg): 指定组件的前景色。

- height: 指定组件的高度。
- image: 指定组件中的图片。
- justify: 指定组件中多行文本的对齐方式。
- text: 指定组件中的文本, 可以使用 “\n” 表示换行。
- value: 指定组件被选中后关联变量的值。
- variable: 指定组件所关联的变量。
- width: 指定组件的宽度。

对于单选框和复选框, variable 是比较关键的参数。由 variable 指定的变量应使用 Tkinter.IntVar, 或者 Tkinter.StringVar 生成。其中 Tkinter.IntVar 生成一个整型变量, 而 Tkinter.StringVar 将生成一个字符串变量。

当使用 Tkinter.IntVar 或者 Tkinter.StringVar 生成变量后, 可以使用 set 方法设置变量的初始值。如果该初始值与组件的 value 所指定的值相等, 则该组件处于被选中状态。如果其他组件被选中, 则变量值将被更改为该组件 value 所指定的值。如下所示的 TkinterCheck.py 创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: TkinterCheck.py
#
import Tkinter
root = Tkinter.Tk()
r = Tkinter.StringVar()
r.set('1')
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '1',
                             text = 'Radio1')
radio.pack()
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '2',
                             text = 'Radio2' )
radio.pack()
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '3',
                             text = 'Radio3' )
radio.pack()
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '4',
                             text = 'Radio4' )
radio.pack()
c = Tkinter.IntVar()
```

导入 Tkinter 模块

使用 StringVar 生成字符串变量用于单选框组件

初始化变量值

生成单选框组件

设置单选框关联的变量

设置选中单选框时其所关联的变量的值, 即 r 的值

设置单选框显示的文本

当选中该单选框时, r 的值为 2

当选中该单选框时, r 的值为 3

当选中该单选框时, r 的值为 4

使用 IntVar 生成整型变量用于复选框


```

c.set(1)
check = Tkinter.Checkbutton(root,
                             text = 'Checkbutton', # 设置复选框的文本
                             variable = c,         # 设置复选框关联的变量
                             onvalue = 1,           # 当选中复选框时, c 的值为 1
                             offvalue = 2)          # 当未选中复选框时, c 的值为 2

check.pack()
root.mainloop()
print r.get() # 输出 r 的值
print c.get() # 输出 c 的值

```

运行 TkinterCheck.py 脚本后将创建如图 12-10 所示的单选框和复选框。

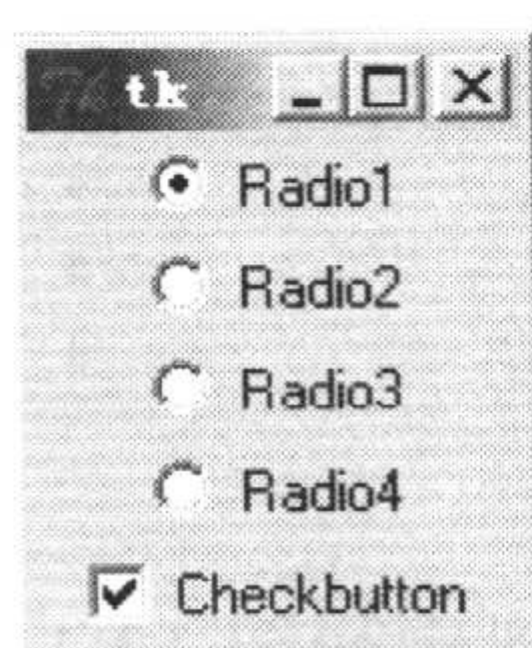


图 12-10 创建单选框和复选框

对于单选框组件和复选框组件还有一个比较特殊的控制参数 `indicatoron`, 当向其传递值 0 时, 组件将被绘制成按钮的形式, 被选中的组件处于按下状态。如下所示的 TkinterRCButton.py 脚本创建按钮形式的单选框和复选框。

```

# -*- coding:utf-8 -*-
# file: TkinterRCButton.py
#
import Tkinter # 导入 Tkinter 模块
root = Tkinter.Tk()
r = Tkinter.StringVar() # 使用 StringVar 生成字符串变量用于单选框组件
r.set('1') # 初始化变量值
radio = Tkinter.Radiobutton(root, # 生成单选框组件
                             variable = r, # 设置单选框关联的变量
                             value = '1', # 设置选中单选框时其所关联的变量的值, 即 r 的值
                             indicatoron = 0, # 将单选框绘制成按钮样式
                             text = 'Radio1') # 设置单选框显示的文本

radio.pack()
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '2', # 当选中该单选框时, r 的值为 2
                             indicatoron = 0,
                             text = 'Radio2' )

radio.pack()
radio = Tkinter.Radiobutton(root,
                             variable = r,
                             value = '3', # 当选中该单选框时, r 的值为 3

```



```

        indicatoron = 0,
        text = 'Radio3' )

radio.pack()
radio = Tkinter.Radiobutton(root,
        variable = r,
        value = '4',           # 当选中该单选框时, r 的值为 4
        indicatoron = 0,
        text = 'Radio4' )

radio.pack()
c = Tkinter.IntVar()           # 使用 IntVar 生成整型变量用于复选框
c.set(1)
check = Tkinter.Checkbutton(root,
        text = 'Checkbutton',  # 设置复选框的文本
        variable = c,          # 设置复选框关联的变量
        indicatoron = 0,       # 将复选框绘制成按钮样式
        onvalue = 1,           # 当选中复选框时, c 的值为 1
        offvalue = 2)          # 当未选中复选框时, c 的值为 2

check.pack()
root.mainloop()

```

运行 TkinterRCButton.py 脚本后将创建如图 12-11 所示的单选框和复选框。

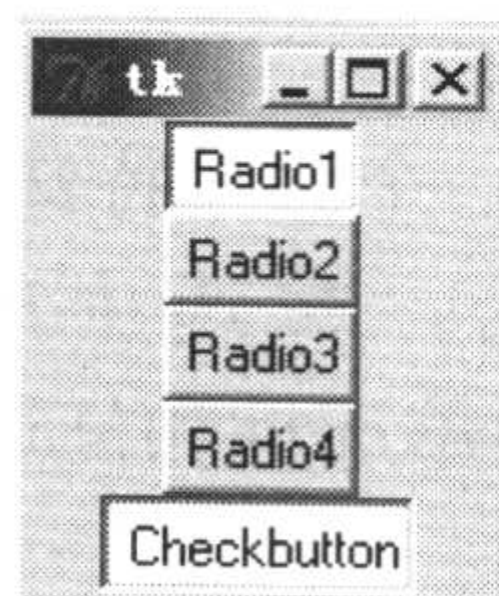


图 12-11 按钮样式的单选框和复选框

12.2.8 绘制图形

使用 Tkinter.Canvas 创建 Canvas 绘图组件后, 可以使用 Canvas 提供的方法在 Canvas 组件中绘制直线、圆弧、矩形以及图片等。Canvas 绘图组件的控制参数主要有以下几个。

- background (bg): 指定绘图组件的背景色。
- borderwidth (bd): 指定绘图组件的边框宽度。
- bitmap: 指定绘图组件中的位图。
- foreground (fg): 指定绘图组件的前景色。
- height: 指定绘图组件的高度。
- image: 指定绘图组件中的图片。
- width: 指定绘图组件的宽度。

Canvas 绘图组件的绘图方法主要有以下几种。

- create_arc: 绘制圆弧。
- create_bitmap: 绘制位图, 支持 XBM。
- create_image: 绘制图片, 支持 GIF。
- create_line: 绘制直线。
- create_oval: 绘制椭圆。
- create_polygon: 绘制多边形。
- create_rectangle: 绘制矩形。
- create_text: 绘制文字。
- create_window: 绘制窗口。
- delete: 删除绘制的图形。

如下所示的 TkinterCanvas.py 使用 Canvas 绘图组件的绘图方法绘制了不同的图形。

```
# -*- coding:utf-8 -*-
# file: TkinterCanvas.py
#
import Tkinter                                     # 导入 Tkinter 模块
root = Tkinter.Tk()
canvas = Tkinter.Canvas(root,
                           width = 600,
                           height = 480,
                           bg = 'white')
im = Tkinter.PhotoImage(file='python.gif')
canvas.create_image(300,50,image = im)

canvas.create_text(302,77,
                  text = 'Use Canvas'
                  ,fill = 'gray')
canvas.create_text(300,75,
                  text = 'Use Canvas',
                  fill = 'blue')
canvas.create_polygon(290,114,316,114,330,130,
                     310,146,284,146,270,130)
canvas.create_oval(280,120,320,140,
                  fill = 'white')
canvas.create_line(250,130,350,130)
canvas.create_line(300,100,300,160)
canvas.create_rectangle(90,190,510,410,
                        width=5)
canvas.create_arc(100, 200, 500, 400,
                  start=0, extent=240,
                  fill="pink")
canvas.create_arc(103,203,500,400,
                  start=241, extent=118,
                  fill="red")

# 指定 Canvas 组件的宽度
# 指定 Canvas 组件的高度
# 指定 Canvas 组件的背景色
# 使用 PhotoImage 打开图片
# 使用 create_image 将图片添加到 Canvas 组件中
# 使用 create_text 方法绘制文字
# 所绘制文字的内容
# 所绘制文字的颜色为灰色

# 使用 create_polygon 方法绘制六边形

# 使用 create_oval 方法绘制椭圆
# 设置椭圆用白色填充
# 使用 create_line 绘制直线

# 使用 create_rectangle 绘制一个矩形
# 设置矩形线宽为 5 个像素
# 使用 create_arc 绘制圆弧
# 设置圆弧的起止角度
# 设置圆弧填充颜色
```



```
canvas.pack()  
root.mainloop()
```

将 Canvas 添加到主窗口

运行 TkinterCanvas.py 脚本后将创建如图 12-12 所示的窗口。

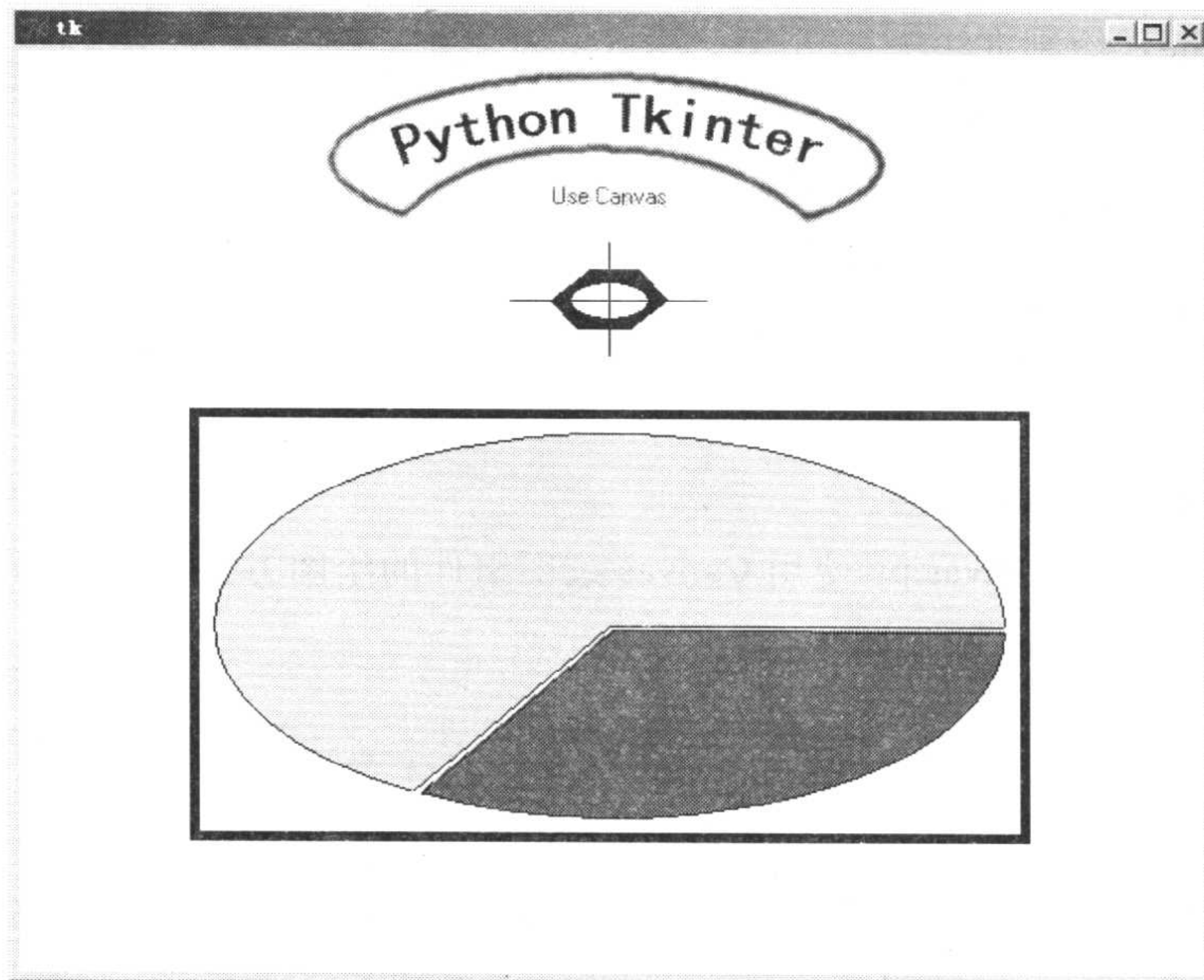


图 12-12 使用 Canvas 绘制图形

12.3 事件处理

Tkinter 中的事件相当于上一章中 MFC 中的消息。对于按钮组件、菜单组件等可以在创建组件时通过 `command` 参数指定其事件的处理函数。除了组件所触发的事件外，在创建右键弹出菜单时，还处理了右击事件。类似的事件可以归结为鼠标事件、键盘事件和窗口事件。

12.3.1 事件表示

鼠标事件主要指鼠标按键的按下、释放，鼠标滚轮的滚动，鼠标指针移进、移出组件等所触发的事件。键盘事件主要指键的按下、释放等所触发的事件。窗口事件是指改变窗口大小、组件状态等变化所触发的事件。

对于鼠标事件、键盘事件和窗口事件可以采用事件绑定的方法确定消息的处理方式。事件绑定可以使用组件的 `bind` 方法进行实例绑定，或者使用 `bind_class` 方法进行类绑定，分别调用函数或者类来响应事件。`bind_all` 方法也可以用来绑定事件，`bind_all` 方法将所有组件事件绑定到事件响应函数上。这 3 种方法的原型分别如下所示。


```
bind(sequence, func, add)
bind_class(className, sequence, func, add)
bind_all(sequence, func, add)
```

各参数含义如下所示。

- sequence: 所绑定的事件。
- func: 所绑定的事件处理函数。
- add: 可选参数, 为空字符或者“+”。
- className: 所绑定的类。

其中, sequence 表示所绑定的事件, 其必须为以“<>”包围的字符串。对于鼠标事件可以使用以下几种表示方式。

- <Button-1>: 表示鼠标左键按下, 而<Button-2>表示中键, <Button-3>表示右键。
- <ButtonPress-1>: 表示鼠标左键按下, 与<Button-1>相同。
- <ButtonRelease-1>: 表示鼠标左键释放。
- <B1-Motion>: 表示按住鼠标左键移动。
- <Double-Button-1>: 表示双击鼠标左键。
- <Enter>: 表示鼠标指针进入某一组件区域。
- <Leave>: 表示鼠标指针离开某一组件区域。
- <MouseWheel>: 表示鼠标滚轮动作。

上述的鼠标事件中凡数字均可替换成 2 或 3。其中 2 表示鼠标中键, 3 表示鼠标右键。例如<B3-Motion>表示按住鼠标右键移动, <Double-Button-2>表示双击鼠标中键等。对于键盘事件可以使用以下几种表示方式。

- <KeyPress-A>: 表示按下 A 键, 可用其他键代替。
- <Alt-KeyPress-A>: 表示同时按下 Alt 键和 A 键。
- <Ctrl-KeyPress-A>: 表示同时按下 Ctrl 键和 A 键。
- <Shift-KeyPress-A>: 表示同时按下 Shift 键和 A 键。
- <Double-KeyPress-A>: 表示快速地按两下 A 键。
- <Lock-KeyPress-A>: 表示 Caps Lock 键打开后按下 A 键。

上述的键盘事件还可以使用 Alt、Ctrl 和 Shift 组合。例如, <Alt-Ctrl-Shift-KeyPress-B>表示同时按下 Alt、Ctrl、Shift 键和 B 键。其中, KeyPress 可以用 KeyRelease 替换, 表示当按键释放时触发事件。需要注意的是字母区分大小写, 如果使用<KeyPress-A>, 则只有按下 Shift 键或者 Caps Lock 键打开时才触发事件。对于窗口事件可以使用以下几种表示方法。

- Activate: 当组件由不可用转为可用时触发。

- Configure: 当组件大小改变时触发。
- Deactivate: 当组件由可用转为不可用时触发。
- Destroy: 当组件被销毁时触发。
- Expose: 当组件从被遮挡状态中暴露出来时触发。
- FocusIn: 当组件获得焦点时触发。
- FocusOut: 当组件失去焦点时触发。
- Map: 当组件由隐藏状态变为显示状态时触发。
- Property: 当窗体的属性被删除或改变时触发。
- Unmap: 当组件由显示状态变为隐藏状态时触发。
- Visibility: 当组件变为可视状态时触发。

12.3.2 响应事件

窗体中的事件被绑定到函数后, 当该事件被触发后将调用所绑定的函数进行处理。事件触发后系统将向该函数传递一个 event 对象的参数, 因此被绑定的响应事件的函数应该定义成如下所示的形式。

```
def function(event):
    <语句>
```

其中 event 对象具有以下属性。

- char: 按键字符, 仅对键盘事件有效。
- keycode: 按键名, 仅对键盘事件有效。
- keysym: 按键编码, 仅对键盘事件有效。
- num: 鼠标按键, 仅对鼠标事件有效。
- type: 所触发的事件类型。
- widget: 引起事件的组件。
- width, height: 组件改变后的大小, 仅对 Configure 有效。
- x, y: 鼠标当前位置, 相对于窗口。
- x_root, y_root: 鼠标当前位置, 相对于整个屏幕。

如下所示的 TkinterDraw.py 使用事件处理创建一个简单的绘图程序。在脚本中主要使用了 events 对象的 x 和 y 属性用于绘制图形。

```
# -*- coding:utf-8 -*-
# file: TkinterDraw.py
#
import Tkinter
class MyButton:
# 导入 Tkinter 模块
# 定义按钮类
```

第12章 使用Tkinter 编写GUI

```

def __init__(self, root, canvas, label, type):
    self.root = root
    self.canvas = canvas
    self.label = label
    if type == 0:
        button = Tkinter.Button(root, text = 'DrawLine',
                                command = self.DrawLine)
    elif type == 1:
        button = Tkinter.Button(root, text = 'DrawArc',
                                command = self.DrawArc)
    elif type == 2:
        button = Tkinter.Button(root, text = 'DrawRec',
                                command = self.DrawRec)
    else :
        button = Tkinter.Button(root, text = 'DrawOval',
                                command = self.DrawOval)
    button.pack(side = 'left')
def DrawLine(self):
    self.label.text.set('Draw Line')
    self.canvas.SetStatus(0)
def DrawArc(self):
    self.label.text.set('Draw Arc')
    self.canvas.SetStatus(1)
def DrawRec(self):
    self.label.text.set('Draw Rectangle')
    self.canvas.SetStatus(2)
def DrawOval(self):
    self.label.text.set('Draw Oval')
    self.canvas.SetStatus(3)
class MyCanvas:
    def __init__(self, root):
        self.status = 0
        self.draw = 0
        self.root = root
        self.canvas = Tkinter.Canvas(root, bg = 'white',
                                     width = 600,
                                     height = 480)
        self.canvas.pack()
        self.canvas.bind('<ButtonRelease-1>', self.Draw)
        self.canvas.bind('<Button-2>', self.Exit)
        self.canvas.bind('<Button-3>', self.Del)
        self.canvas.bind_all('<Delete>', self.Del)
        self.canvas.bind_all('<KeyPress-d>', self.Del)
        self.canvas.bind_all('<KeyPress-e>', self.Exit)
    def Draw(self, event):
        if self.draw == 0:
            self.x = event.x
            self.y = event.y
            self.draw = 1

```

类初始化
保存引用值
根据类型创建按钮
DrawLine 按钮事件处理函数
DrawArc 按钮事件处理函数
DrawRec 按钮事件处理函数
DrawOval 按钮事件处理函数
定义 Canvas 类
保存引用值
生成 Canvas 组件
绑定事件到左键
绑定事件到中键
绑定事件到右键
绑定事件到 Delete 键
绑定事件到 D 键
绑定事件到 E 键
绘图事件处理函数
判断是否绘图


```

else:
    if self.status == 0:
        self.canvas.create_line(self.x,self.y,
                                event.x,event.y)
        self.draw = 0
    elif self.status == 1:
        self.canvas.create_arc(self.x,self.y,
                                event.x,event.y)
        self.draw = 0
    elif self.status == 2:
        self.canvas.create_rectangle(self.x,self.y,
                                      event.x,event.y)
        self.draw = 0
    else:
        self.canvas.create_oval(self.x,self.y,
                                 event.x,event.y)
        self.draw = 0
def Del(self,event):
    items = self.canvas.find_all()
    for item in items:
        self.canvas.delete(item)
def Exit(self,event):
    self.root.quit()
def SetStatus(self,status):
    self.status = status
class MyLabel:
    def _init_(self,root):
        self.root = root
        self.canvas = canvas
        self.text = Tkinter.StringVar()
        self.text.set('Draw Line')
        self.label = Tkinter.Label(root,textvariable = self.text,
                                    fg = 'red',width = 50)
        self.label.pack(side = 'left')
root = Tkinter.Tk()
canvas = MyCanvas(root)
label = MyLabel(root)
MyButton(root,canvas,label,0)
MyButton(root,canvas,label,1)
MyButton(root,canvas,label,2)
MyButton(root,canvas,label,3)
root.mainloop()

```

根据 self.status 绘制不同图形

当按下右键或 D 键时删除图形

当按下中键或 E 键时退出

设置绘制的图形

定义标签类

类初始化

保存引用

生成标签引用变量

生成标签

生成主窗口

生成绘图组件

生成标签

生成按钮

进入消息循环

运行 TkinterDraw.py 脚本后将创建如图 12-13 所示的窗口。可以单击按钮选择要绘制的图形。在窗口中单击鼠标左键，然后移动到另一位置再单击左键将绘制图形。单击右键或者按键盘上的 D 键将删除多绘制的图形。单击鼠标中键或者按键盘上的 E 键将关闭窗口。

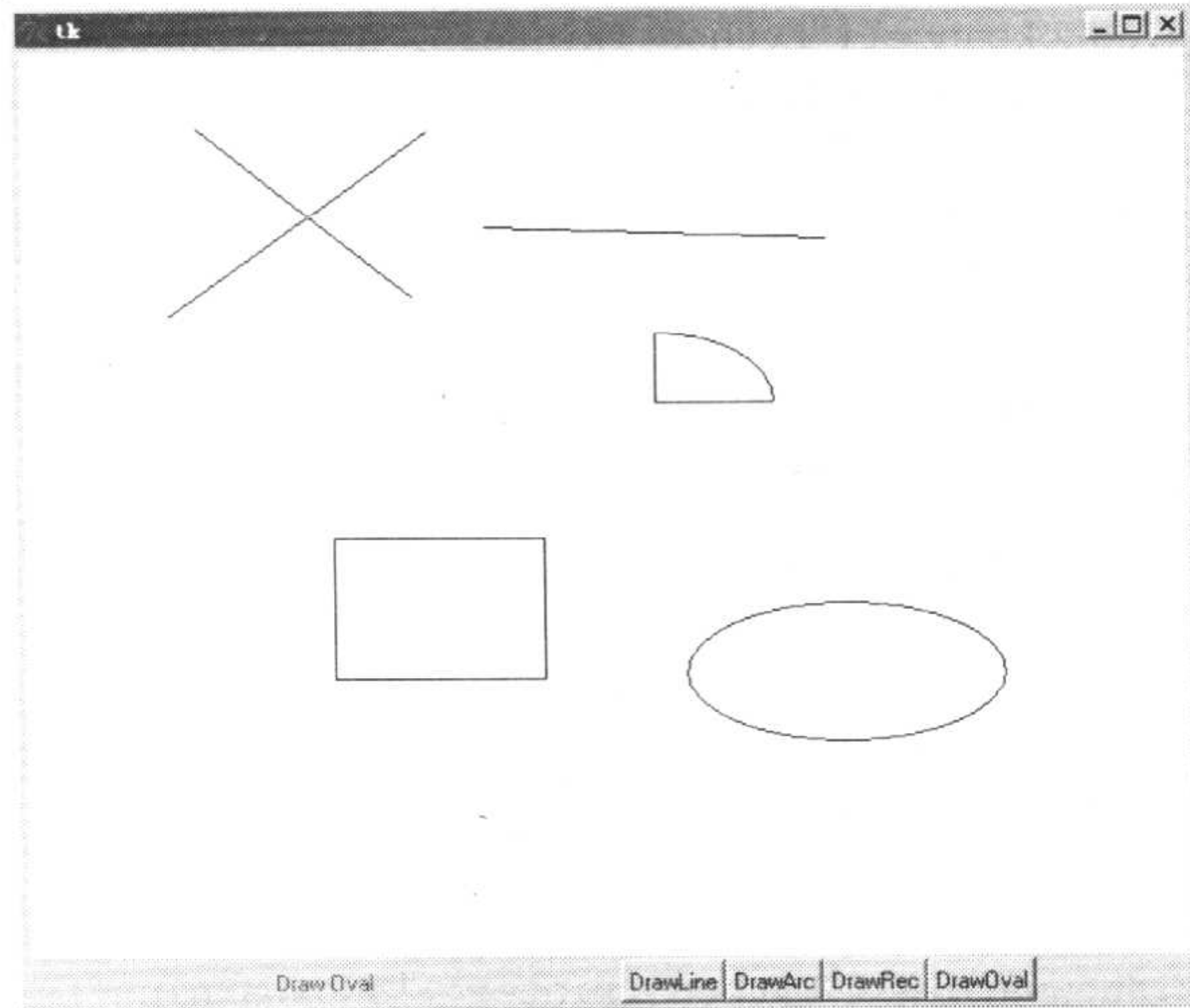


图 12-13 简单的绘图程序

12.4 创建对话框

在 Tkinter 中，提供了标准的对话框。在脚本中可以使用标准对话框与用户交互。如果 Tkinter 提供的对话框不能满足要求，可以使用 Toplevel 来创建对话框。

12.4.1 使用标准对话框

标准对话框包含简单的消息框和用户输入对话框。其中，信息框以窗口的形式向用户输出信息，也可以获取用户所单击的按钮。输入对话框要求用户输入字符串、整型或者浮点型的值。

1. 消息框

Tkinter 中的 tkMessageBox 模块提供了几种简单的消息框。使用 tkMessageBox 模块中的 askokcancel、askquestion、askyesno、showerror、showinfo 和 showwarning 可以创建简单的消息框。使用这些函数时只需向其传递 title 和 message 参数即可。

如下所示的 TkinterMessageBox.py 脚本使用 tkMessageBox 创建简单的消息框。

```
# -*- coding:utf-8 -*-
# file: TkinterMessageBox.py
#
import Tkinter
import tkMessageBox

def cmd():
    global n
    global buttontext
    n = n + 1
    if n == 1:
        # 导入 Tkinter 模块
        # 导入 tkMessageBox 模块
        # 按钮消息处理函数
        # 使用全局变量 n
        # 使用全局变量 buttontext

        # 判断 n 的值，显示不同的消息框
```



```

tkMessageBox.askokcancel('Python Tkinter','askokcancel') # 使用 askokcancel 函数
buttontext.set('skquestion') # 更改按钮上的文字
elif n == 2:
    tkMessageBox.askquestion('Python Tkinter','skquestion') # 使用 askquestion 函数
    buttontext.set('askyesno')
elif n == 3:
    tkMessageBox.askyesno('Python Tkinter','askyesno') # 使用 askyesno 函数
    buttontext.set('showerror')
elif n == 4:
    tkMessageBox.showerror('Python Tkinter','showerror') # 使用 showerror 函数
    buttontext.set('showinfo')
elif n == 5:
    tkMessageBox.showinfo('Python Tkinter','showinfo') # 使用 showinfo 函数
    buttontext.set('showwarning')
else :
    n = 0 # 将 n 赋值为 0 重新开始循环
    tkMessageBox.showwarning('Python Tkinter','showwarning') # 使用 showwarning 函数
    buttontext.set('askokcancel')
n = 0 # 为 n 赋初始值
root = Tkinter.Tk()
buttontext = Tkinter.StringVar() # 生成关联按钮文字的变量
buttontext.set('askokcancel') # 设置 buttontext 值
button = Tkinter.Button(root, # 生成按钮
    textvariable = buttontext, # 设置关联变量
    command = cmd) # 设置事件处理函数
button.pack()
root.mainloop() # 进入消息循环

```

运行 TkinterMessageBox.py 脚本后，单击主窗口中的按钮将依次创建如图 12-14～图 12-19 所示的消息框。



图 12-14 askokcancel 消息框



图 12-15 askquestion 消息框



图 12-16 askyesno 消息框

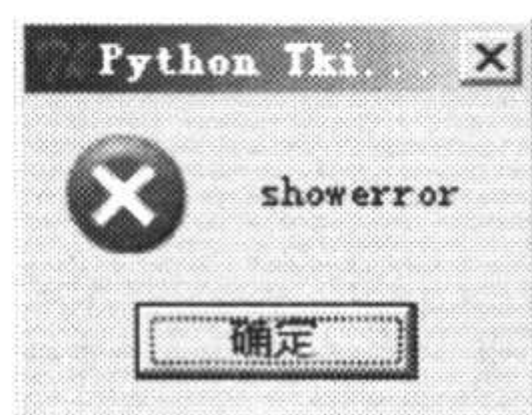


图 12-17 showerror 消息框



图 12-18 showinfo 消息框图

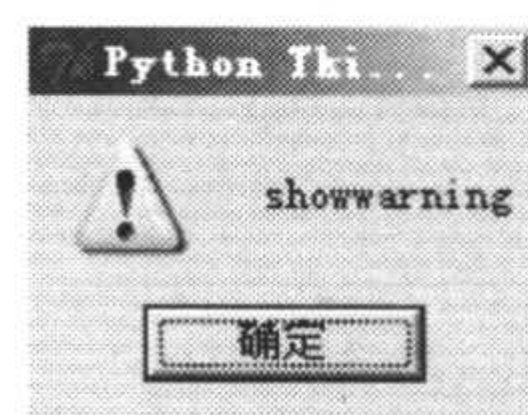


图 12-19 showwarning 消息框

除了上述 6 个标准的消息框以外，还可以使用 tkMessageBox._show 函数创建其他类型的消息框。tkMessageBox._show 函数有以下几个控制参数。

- default: 指定消息框的按钮。

- icon: 指定消息框的图标。
- message: 指定消息框所显示的消息。
- parent: 指定消息框的父组件。
- title: 指定消息框的标题。
- type: 指定消息框的类型。

2. 使用标准对话框

使用 tkSimpleDialog 模块、tkFileDialog 模块和 tkColorChooser 模块可以创建标准的对话框。其中，tkSimpleDialog 模块可以创建标准的输入对话框。tkFileDialog 模块可以创建打开文件和保存文件对话框。tkColorChooser 模块可以创建颜色选择对话框。

tkSimpleDialog 模块可以创建 3 种类型的对话框：输入字符串、输入整数和输入浮点型的对话框。其对应的函数分别为 askstring、askinteger 和 askfloat 函数。其具有以下几个相同的可选参数。

- title: 指定对话框标题。
- prompt: 指定对话框中显示的文字。
- initialvalue: 指定输入框的初始值。

使用 tkSimpleDialog 模块中的函数创建对话框后，将返回对话框中文本框的值。如下所示的脚本 TkinterSimpleDialog.py，分别创建了 3 种简单的输入对话框。

```
# -*- coding:utf-8 -*-
# file: TkinterSimpleDialog.py
#
import Tkinter
import tkSimpleDialog

def InStr():
    r = tkSimpleDialog.askstring('Python Tkinter',
                                'Input String',
                                initialvalue='Tkinter')
    print r
def InInt():
    r = tkSimpleDialog.askinteger('Python Tkinter','Input Integer')
    print r
def InFlo():
    r = tkSimpleDialog.askfloat('Python Tkinter','Input Float')
    print r
root = Tkinter.Tk()
button1 = Tkinter.Button(root,text = 'Input String',
                          command = InStr)
button1.pack(side='left')
button2 = Tkinter.Button(root,text = 'Input Integer',
```

```
# 导入 Tkinter 模块
# 导入 tkSimpleDialog 模块
# 按钮事件处理函数
# 创建字符串输入对话框
# 指定提示字符
# 指定初始化文本
# 输出返回值
# 按钮事件处理函数
# 创建整数输入对话框
# 按钮事件处理函数
# 创建浮点数输入对话框
# 创建按钮
# 指定按钮事件处理函数
```

```

        command = InInt)
button2.pack(side='left')
button2 = Tkinter.Button(root, text = 'Input Float',
        command = InFlo)
button2.pack(side='left')
root.mainloop()

```

指定按钮事件处理函数

指定按钮事件处理函数

进入消息循环

运行 TkinterSimpleDialog.py 脚本后将创建如图所示的窗口。分别单击 3 个按钮将创建如图 12-20~图 12-23 所示的输入对话框。

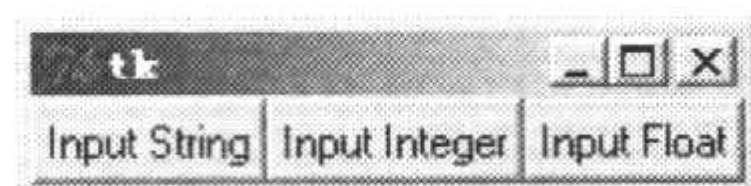


图 12-20 创建的主窗口

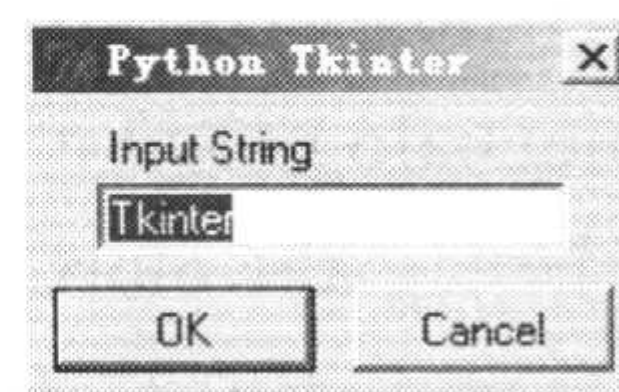


图 12-21 输入字符串窗口

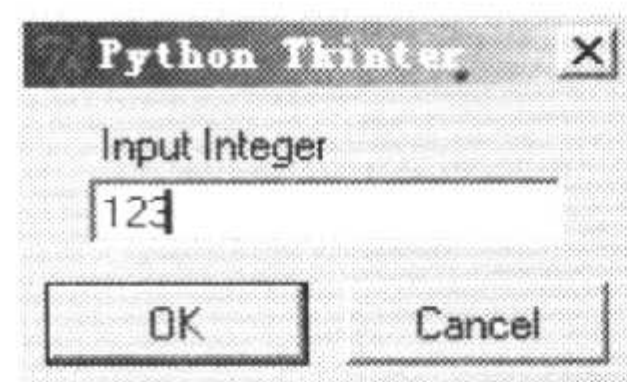


图 12-22 输入整数窗口

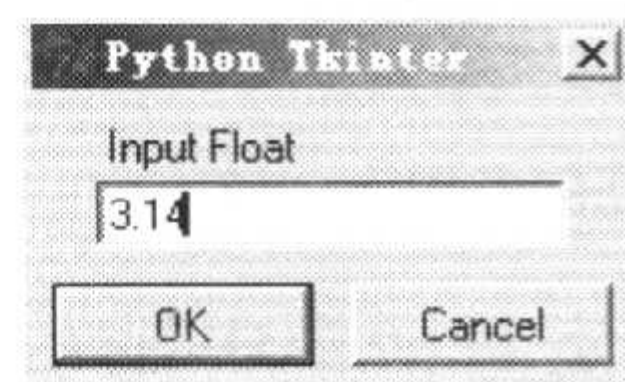


图 12-23 输入浮点数窗口

tkFileDialog 模块中的 askopenfilename 函数可以创建标准的打开文件对话框。asksaveasfilename 可以创建标准的保存文件对话框。其具有以下几个相同的可选参数。

- filetypes: 指定文件类型。
- initialdir: 指定默认目录。
- initialfile: 指定默认文件。
- title: 指定对话框标题。

使用 tkFileDialog 模块中的函数创建对话框后，将返回文件的完整路径。如下所示的脚本 TkinterFileDialog.py 创建了文件打开和保存对话框。

```

# -*- coding:utf-8 -*-
# file: TkinterFileDialog.py
#
import Tkinter
import tkFileDialog
def FileOpen():
    r = tkFileDialog.askopenfilename(title = 'Python Tkinter',
        filetypes=[('Python', '*.py *.pyw'),('All files', '*')])
    print r
def FileSave():
    r = tkFileDialog.asksaveasfilename(title = 'Python Tkinter',

```

导入 Tkinter 模块
导入 tkFileDialog 模块
按钮事件处理函数
创建打开文件对话框
指定文件类型为 Python 脚本
输出返回值
按钮事件处理函数
创建保存文件对话框

第12章 使用Tkinter编写GUI

```

        initialdir=r'E:\Python\code',
        initialfile = 'test.py')

    print r
    root = Tkinter.Tk()
    button1 = Tkinter.Button(root,text = 'File Open',
        command = FileOpen)
    button1.pack(side='left')
    button2 = Tkinter.Button(root,text = 'File Save',
        command = FileSave)
    button2.pack(side='left')
    root.mainloop()

```

指定初始化目录
指定初始化文件
创建按钮
指定按钮事件处理函数
指定按钮事件处理函数
进入消息循环

运行脚本后单击【File Open】按钮，将创建如图 12-24 所示的打开文件对话框。单击【File Save】按钮，将创建如图 12-25 所示的文件保存对话框。

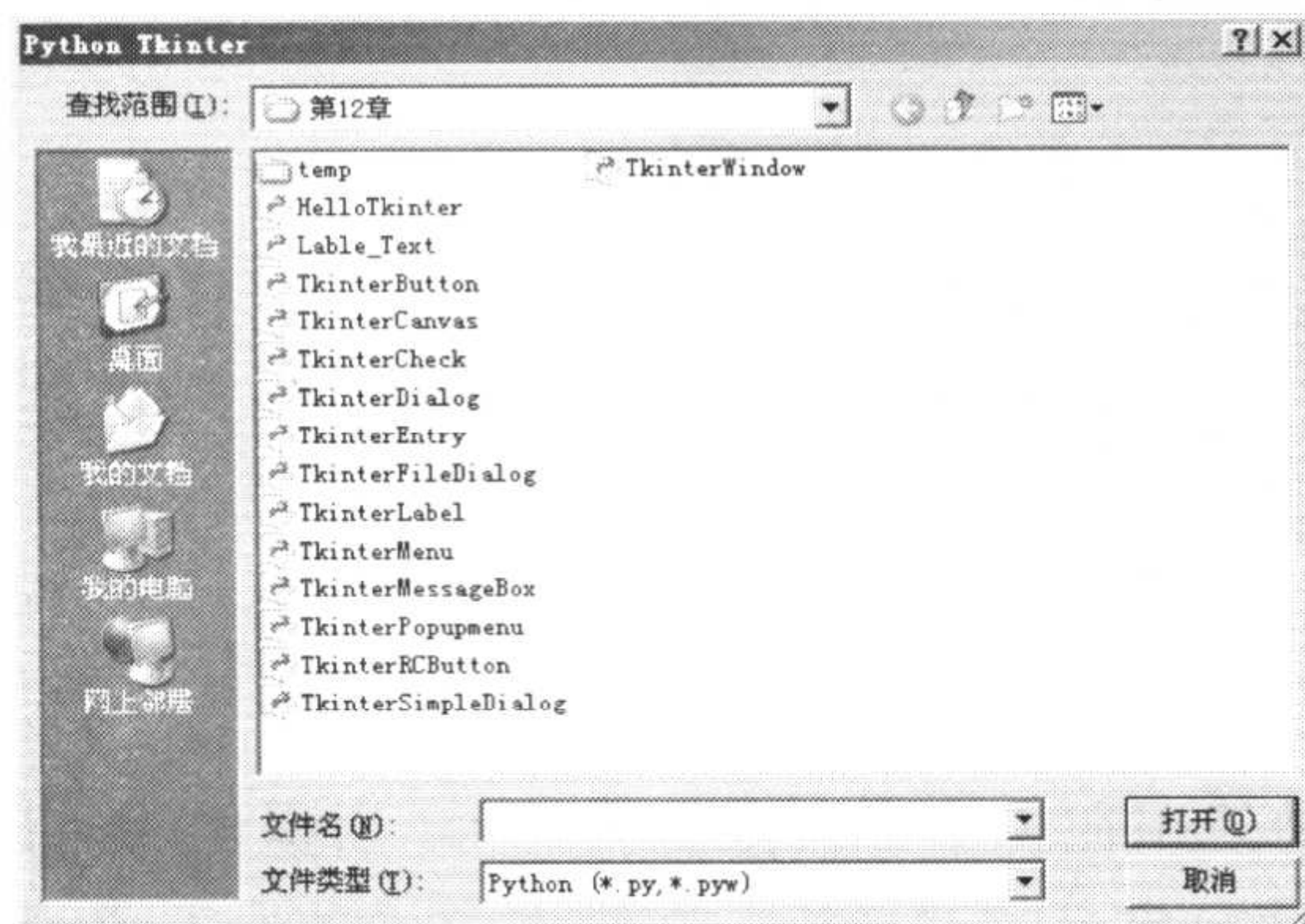


图 12-24 打开文件对话框

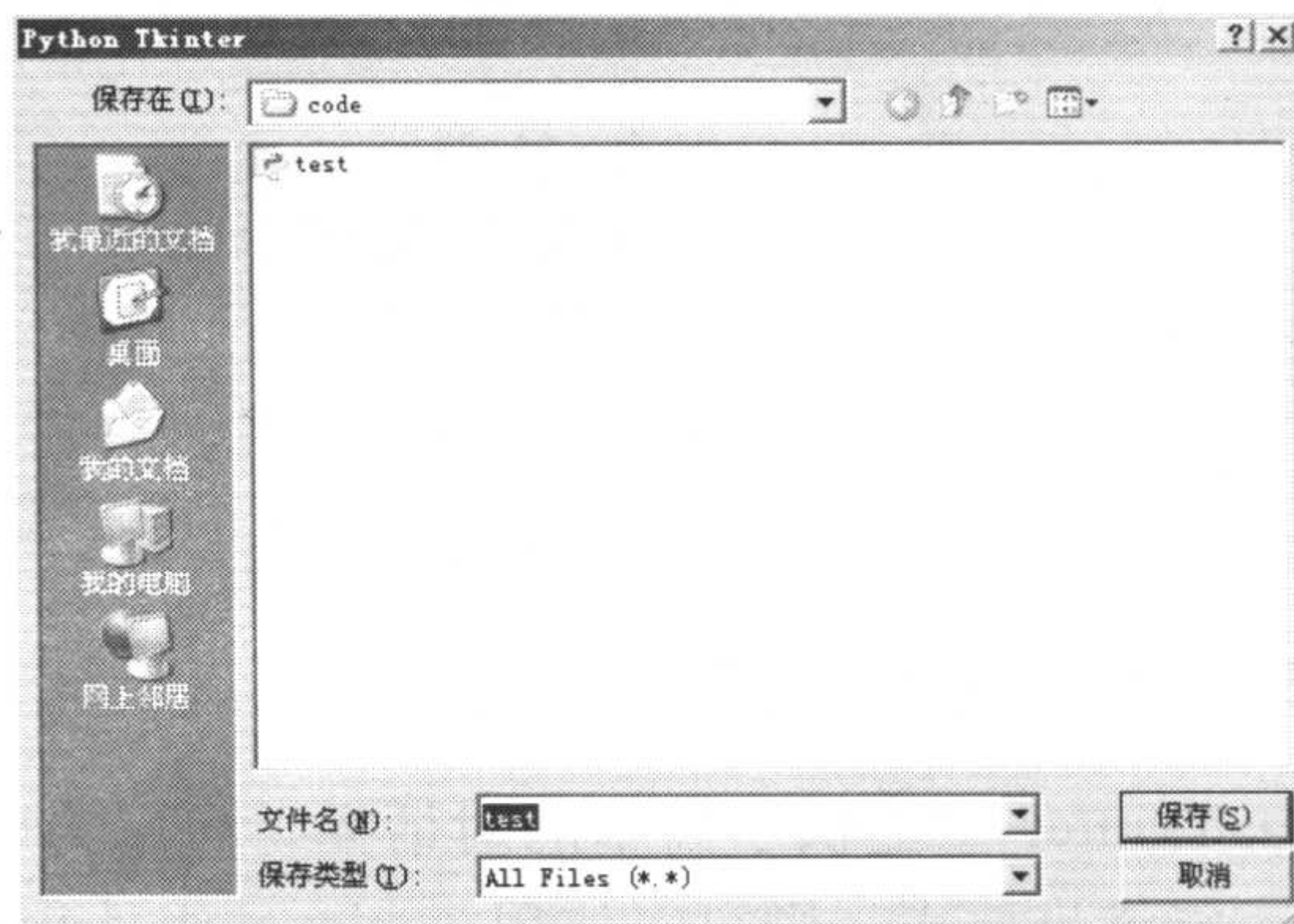


图 12-25 保存文件对话框

tkColorChooser 模块中的 askcolor 函数可以创建标准的颜色选择对话框。其具有以下几个可选参数。

- initialcolor: 指定初始化颜色。
- title: 指定对话框标题。

使用 tkColorChooser 模块中的函数创建对话框后，将返回颜色的 RGB 值以及可以在 Python Tkinter 中使用的颜色字符值。如下所示的脚本 TkinterColorChooser.py 创建了颜色选择对话框。

```

# -*- coding:utf-8 -*-
# file: TkinterColorChooser.py
#
import Tkinter
import tkColorChooser
def ChooseColor():
    r = tkColorChooser.askcolor(title = 'Python Tkinter')
    print r

```

导入 Tkinter 模块
导入 tkColorChooser 模块
按钮事件处理函数
创建颜色选择对话框
输出返回值


```

root = Tkinter.Tk()
button = Tkinter.Button(root, text = 'Choose Color',      # 创建按钮
                        command = ChooseColor)           # 指定按钮事件处理函数
button.pack()
root.mainloop()                                         # 进入消息循环

```

运行 TkinterColorChooser.py 脚本后单击 **【Choose Color】** 命令，将创建如图 12-26 所示的颜色选择对话框。

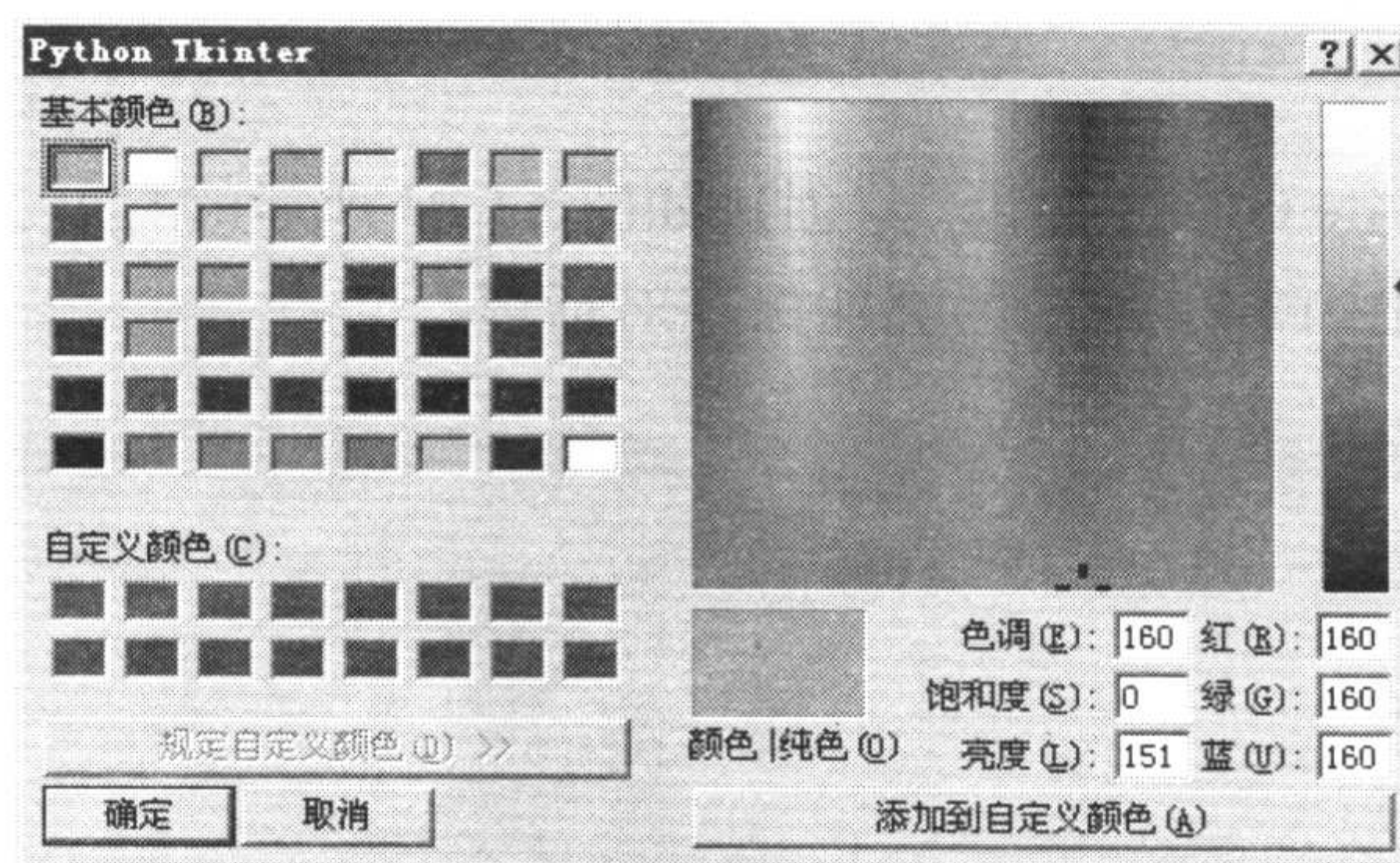


图 12-26 颜色选择对话框

12.4.2 创建自定义对话框

Tkinter 中提供了简单的对话框，可以方便地使用在脚本中。如果 Tkinter 所提供的对话框不能满足要求，则可以使用 Toplevel 组件来创建自定义的对话框。在脚本中可以为 Toplevel 组件添加其他组件，并且定义事件响应函数或者类等。

在使用 Tkinter 创建对话框时，如果对话框中也需要进行事件处理，最好以类的形式来定义对话框，否则可能大量使用全局变量来处理参数。对于代码较多的 Tkinter GUI Python 脚本，整个脚本也应该使用类的方式来组织。

如下所示的 TkinterDialog.py 脚本使用 Toplevel 组件创建一个简单的对话框。

```

# -*- coding:utf-8 -*-
# file: TkinterDialog.py
#
import Tkinter                                     # 导入 Tkinter 模块
import tkMessageBox                               # 导入 tkMessageBox 模块
class MyDialog:                                   # 定义对话框类
    def __init__(self, root):                     # 对话框初始化
        self.top = Tkinter.Toplevel(root)        # 生成 Toplevel 组件
        label = Tkinter.Label(self.top, text='Please Input') # 生成标签组件
        label.pack()
        self.entry = Tkinter.Entry(self.top)     # 生成文本框组件
        self.entry.pack()
        self.entry.focus()                       # 让文本框获得焦点

```


第12章 使用Tkinter编写GUI

```

        button = Tkinter.Button(self.top, text='Ok',      # 生成按钮
                                command=self.Ok)         # 设置按钮事件处理函数
        button.pack()
    def Ok(self):                                         # 定义按钮事件处理函数
        self.input = self.entry.get()                   # 获取文本框中内容, 保存为 input
        self.top.destroy()                              # 销毁对话框
    def get(self):                                        # 返回在文本框中输入的内容
        return self.input
class MyButton():                                       # 定义按钮类
    def __init__(self, root, type):                     # 按钮初始化
        self.root = root                               # 保存父窗口引用
        if type == 0:                                   # 根据类型创建不同按钮
            self.button = Tkinter.Button(root,
                                            text='Create',
                                            command = self.Create) # 设置 Create 按钮的事件处理函数
        else:
            self.button = Tkinter.Button(root,
                                            text='Quit',
                                            command = self.Quit)   # 设置 Quit 按钮的事件处理函数
        self.button.pack()
    def Create(self):                                    # Create 按钮的事件处理函数
        d = MyDialog(self.root)                         # 生成对话框
        self.button.wait_window(d.top)                  # 等待对话框结束
        tkMessageBox.showinfo('Python', 'You input:\n' + d.get()) # 获取对话框中输入值, 并输出
    def Quit(self):                                     # Quit 按钮的事件处理函数
        self.root.quit()                                # 退出主窗口
root = Tkinter.Tk()                                     # 生成主窗口
MyButton(root,0)                                        # 生成 Create 按钮
MyButton(root,1)                                        # 生成 Quit 按钮
root.mainloop()                                        # 进入消息循环

```

运行 TkinterDialog.py 脚本后将创建如图 12-27 所示的窗口, 单击 Create 按钮后将创建如图 12-28 所示的窗口, 单击 **【OK】** 按钮后, 将弹出如图 12-29 所示的信息框。单击 **【Quit】** 按钮, 退出窗口。

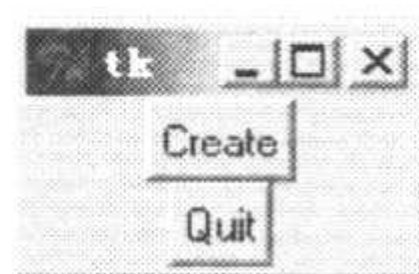


图 12-27 创建的主窗口

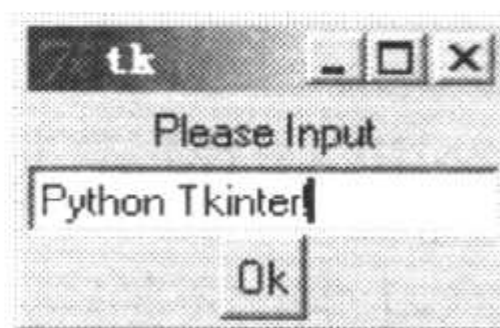


图 12-28 创建的窗口

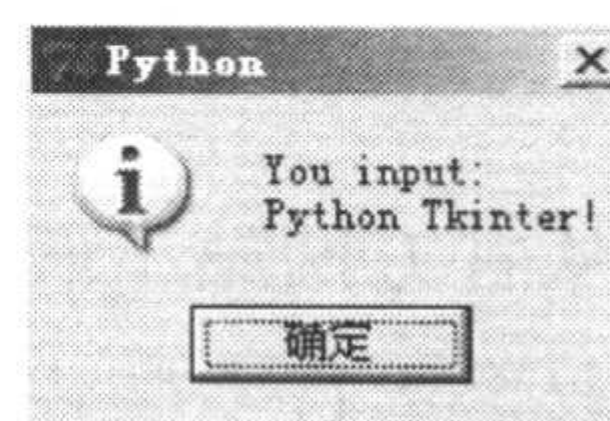


图 12-29 消息框

第 13 章 使用 wxPython 编写 GUI

wxPython 是跨平台 GUI 工具库 wxWidgets 的封装。wxWidgets 库是由 C++ 编写的，它类似于 Windows 的 MFC。wxWidgets 提供了对多种操作系统的支持。由于它具有良好的可移植性，wxPython 也具备了跨平台的能力。在 Python 中使用 wxPython 可以编写具有跨平台能力的 GUI 脚本。

13.1 wxPython 概述

与 Tkinter 不同，wxPython 需要用户自己安装，但安装过程并不复杂，wxPython 的使用也不复杂。有过 MFC 编程经验的用户会很快熟悉 wxPython。

13.1.1 安装 wxPython

由于 wxPython 不是作为 Python 的一部分而随官方发布的，因此需要从其官方网站 <http://www.wxpython.org> 下载安装。根据所安装的 Python 版本下载相应的安装程序。wxPython 安装程序分为 Unicode 版和 Ansi 版。其中 unicode 版提供了对 unicode 的支持，如果在程序中使用需要使用中文，则应该下载相应的 unicode 版的 wxPython。

在 Windows 系统下 wxPython 的安装十分方便。以 Python 2.5 版为例，安装步骤如下所示。

- (1) 从官方网站下载相应的 win32-unicode 版的安装程序。
- (2) 双击运行安装程序，将出现如图 13-1 所示的安装界面。
- (3) 单击 **【Next】** 按钮，将显示如图 13-2 所示的安装协议，选中 **【I accept the agreement】** 单选框。
- (4) 单击 **【Next】** 按钮，将出现如图 13-3 所示的安装路径界面。比较文本框中的路径是否为本机所安装的 Python 路径。如果是，则可以单击 **【Next】** 按钮，进入下一步安装，否则，应将其改成相应的安装路径。
- (5) 剩下的步骤只需单击 **【Next】** 按钮，即可完成安装。

第13章 使用 wxPython 编写 GUI

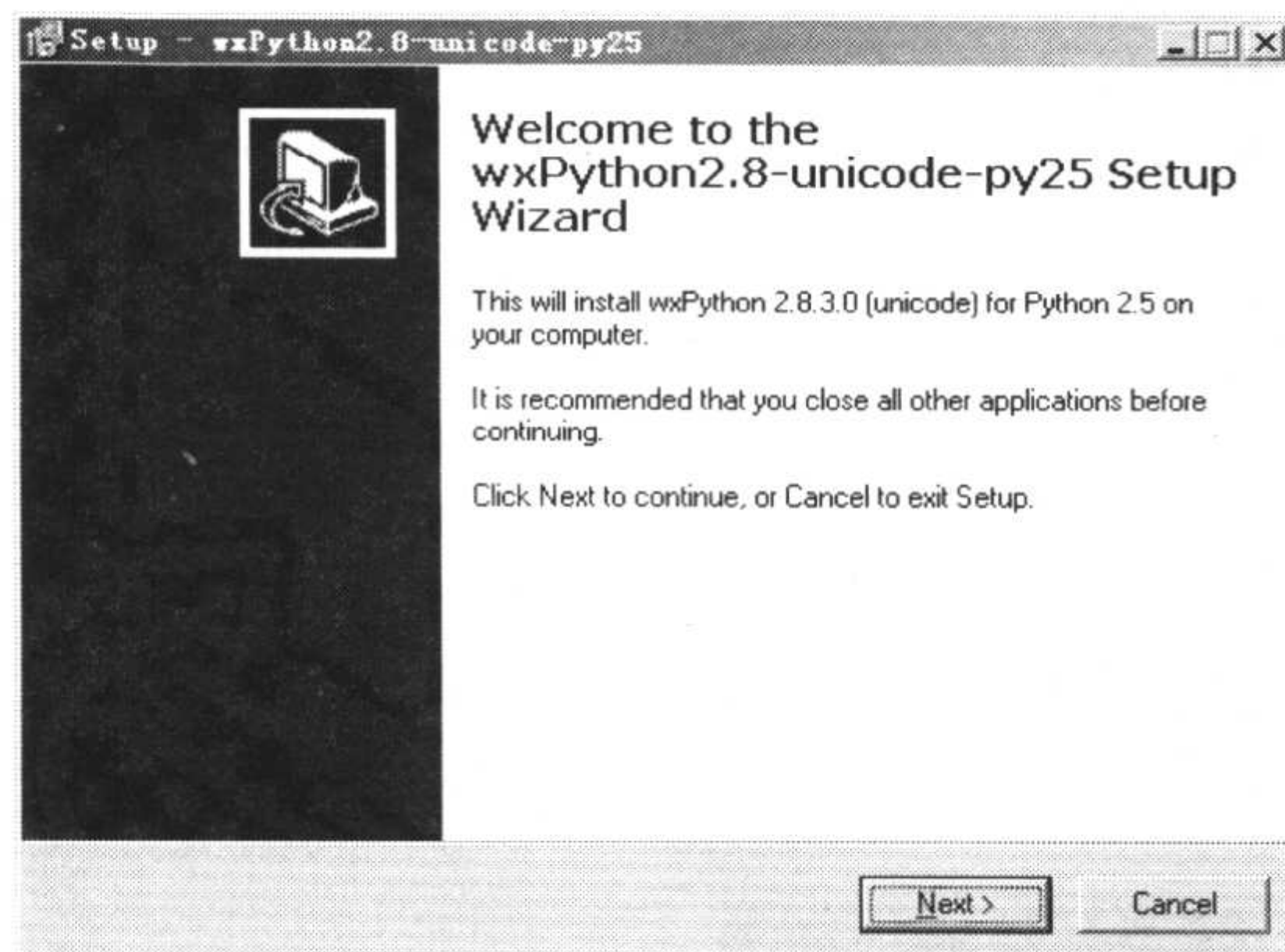


图 13-1 wxPython 安装界面

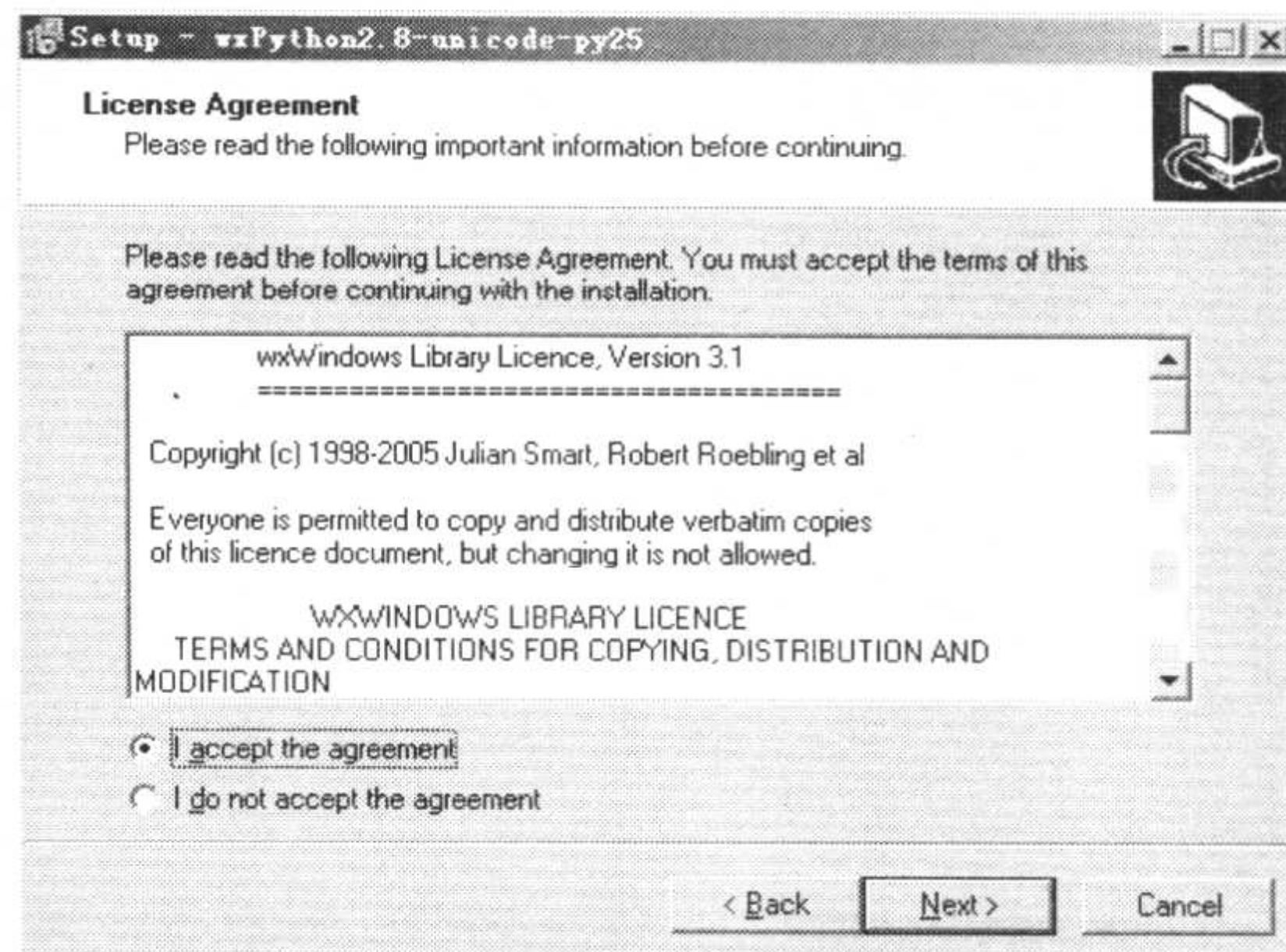


图 13-2 安装协议

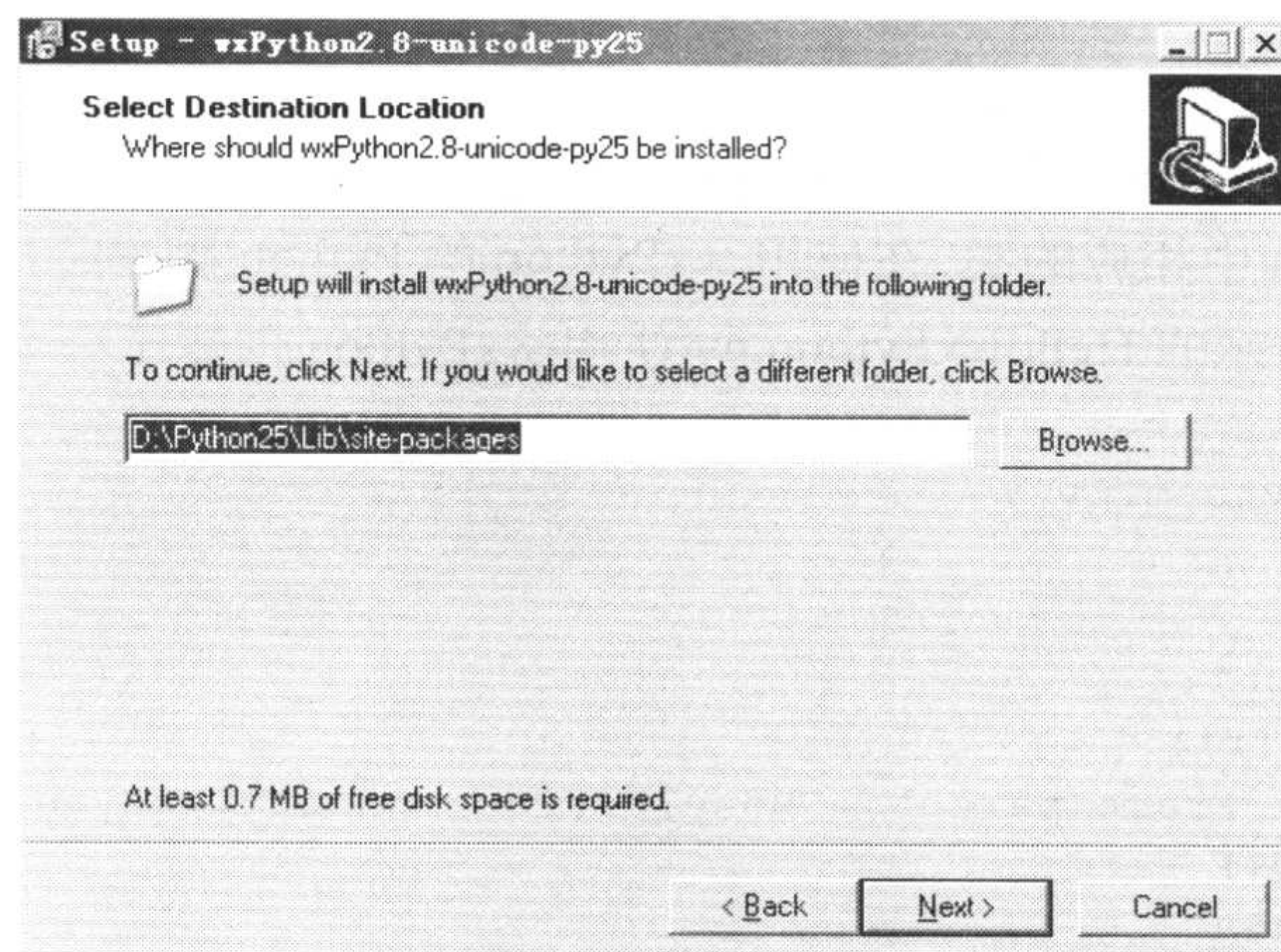


图 13-3 安装路径界面

当安装结束后，可以在 Python 交互式环境下输入以下语句。

```
import wx
```

如果没有出现运行错误，则表示 wxPython 安装成功，可以使用 wxPython 进行 GUI 编程了。除了 wxPython 的安装程序以外，其官方网站还提供了文档和例子的安装文件，该安装文件包含了 wxPython 的参考文档以及一些使用 wxPython 编写的 GUI 脚本。

另外，在 wxPython 的文档和例子安装文件中还提供了 PyShell 和 PyCrust 两个图形交互式 Python Shell，如图 13-4 和图 13-5 所示。

在 wxPython 的文档和例子安装文件中还提供了用于编辑资源文件的 XRCed。wxPython 也使用类似于 MFC 中的资源文件。使用 XRCed 可以方便地生成资源文件，减少代码。

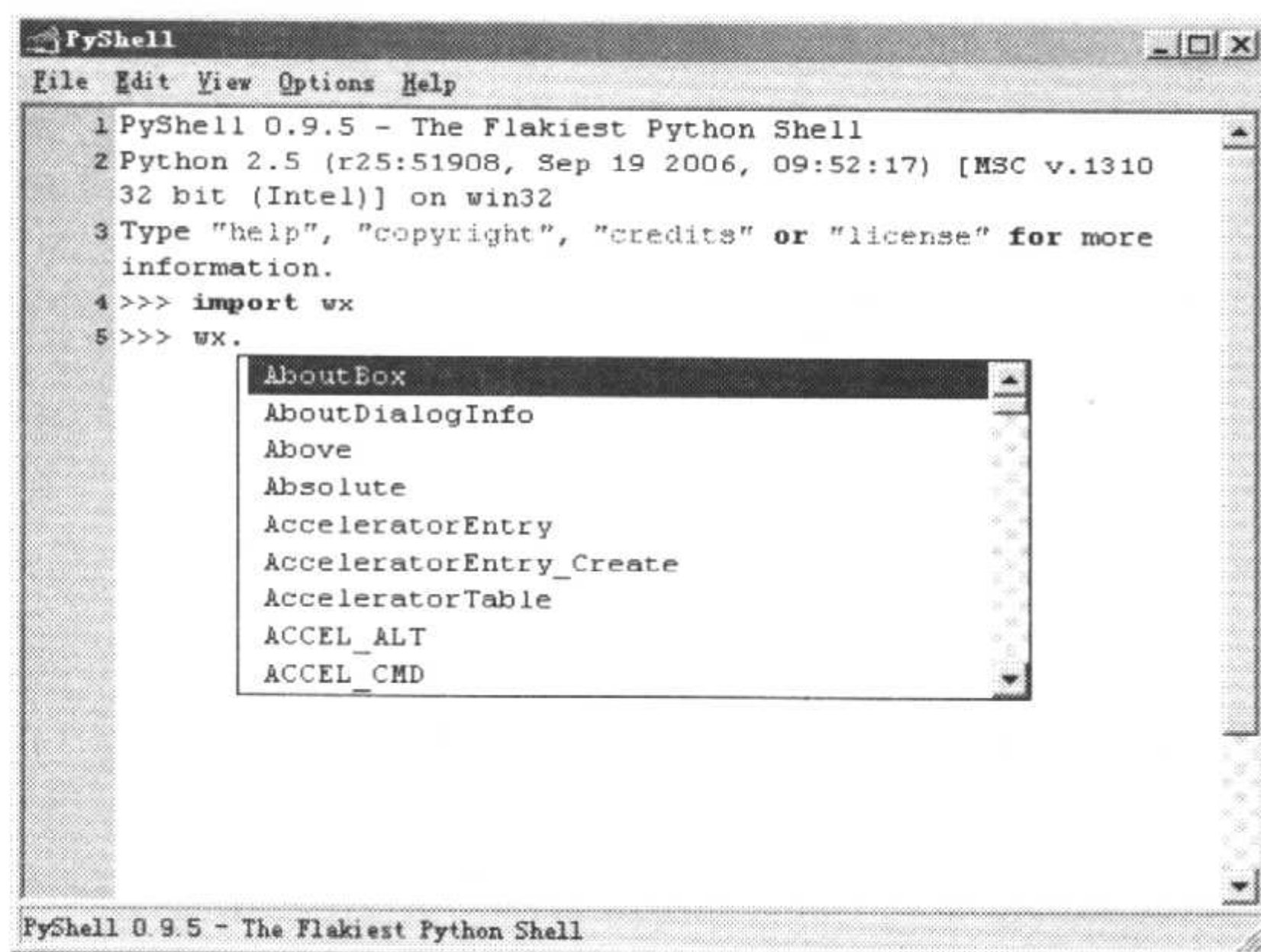


图 13-4 使用 PyShell

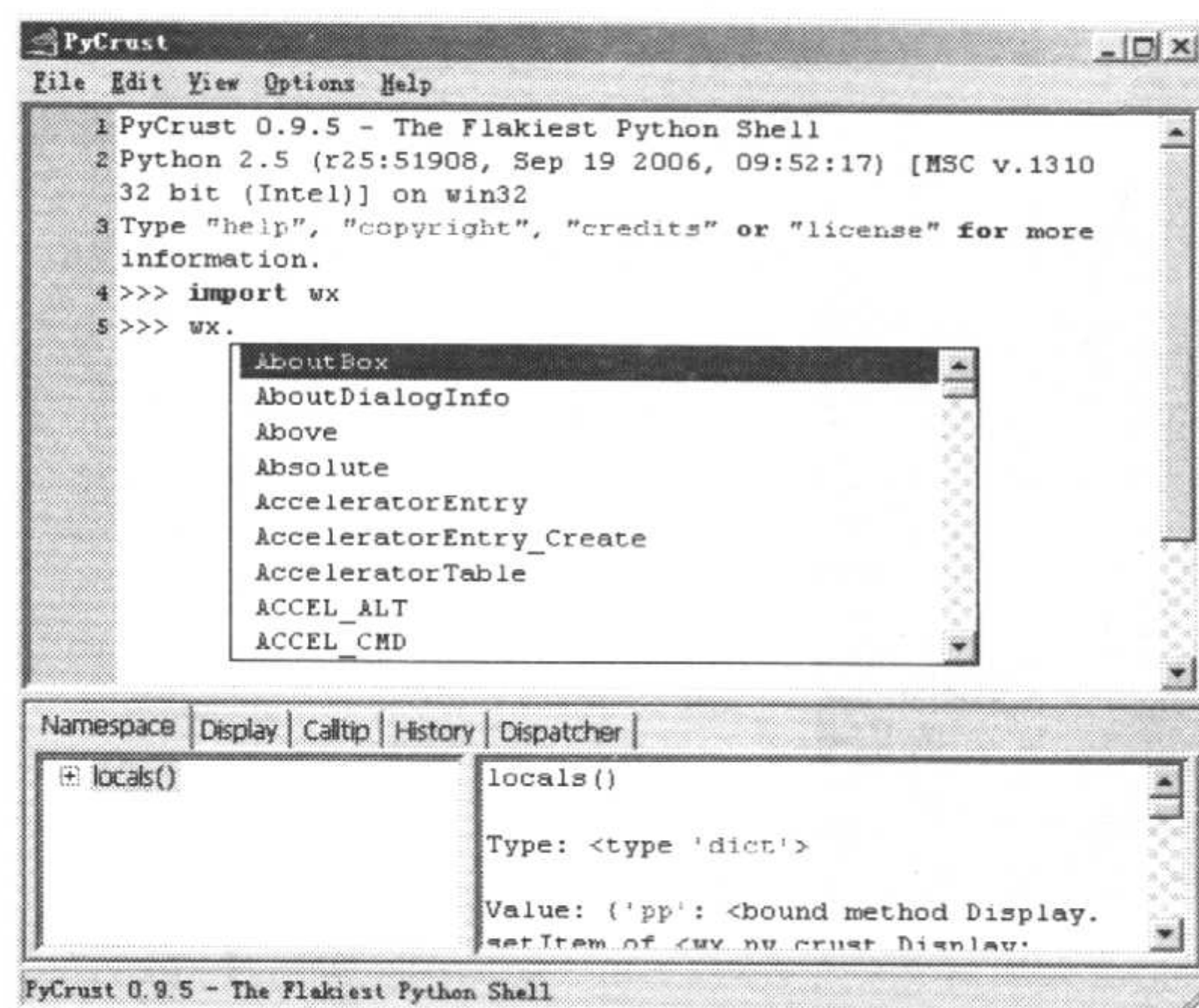


图 13-5 使用 PyCrust

13.1.2 创建窗口

wxPython 是对 wxWidgets C++类库的封装，因此，其使用类似于 Windows 的 MFC。在脚本中应该继承 wxPython 中相应的类。在使用 wxPython 的 Python 脚本时应该首先继承 wxPython 中的 wxApp 类。如下所示的 HellowxPython.py 使用 wxPython 创建了一个简单的窗口。

```
# -*- coding:utf-8 -*-
# file: HellowxPython.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'Hello,wxPython!')
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 HellowxPython.py 脚本后将创建如图 13-6 所示的窗口。

上述的 HellowxPython.py 脚本首先导入 wx 模块，即 wxPython。然后继承 wx 模块中的 App 类创建了一个 MyApp 类。在 MyApp 类中重载了 OnInit 方法。OnInit 方法是窗口初始化的一部分，在该方法中创建了一个窗口框架，并显示该窗口。一般来讲，OnInit 方法最后应返回 True。

在脚本的最后两行，将 MyApp 类实例化成 app 对象，然后调用其 MainLoop 方法进入消息循环。虽然上述脚本代码很少，但即使更复杂的脚本，也遵循这样的过程。复杂的脚本无非是在 OnInit 方法中做更多的初始化工作，定义事件响应函数等，最终还是要进入消息循环的。如下所示的 SimpleHello.py，以更加简洁的形式创建了一个窗口。

```
# -*- coding:utf-8 -*-
```



```
# file: SimpleHello.py
#
import wx
app = wx.PySimpleApp()
frame = wx.Frame(parent = None, title = 'Simple Hello')
frame.Show(True)
app.MainLoop()
```

运行 SimpleHello.py 脚本后将创建如图 13-7 所示的窗口。

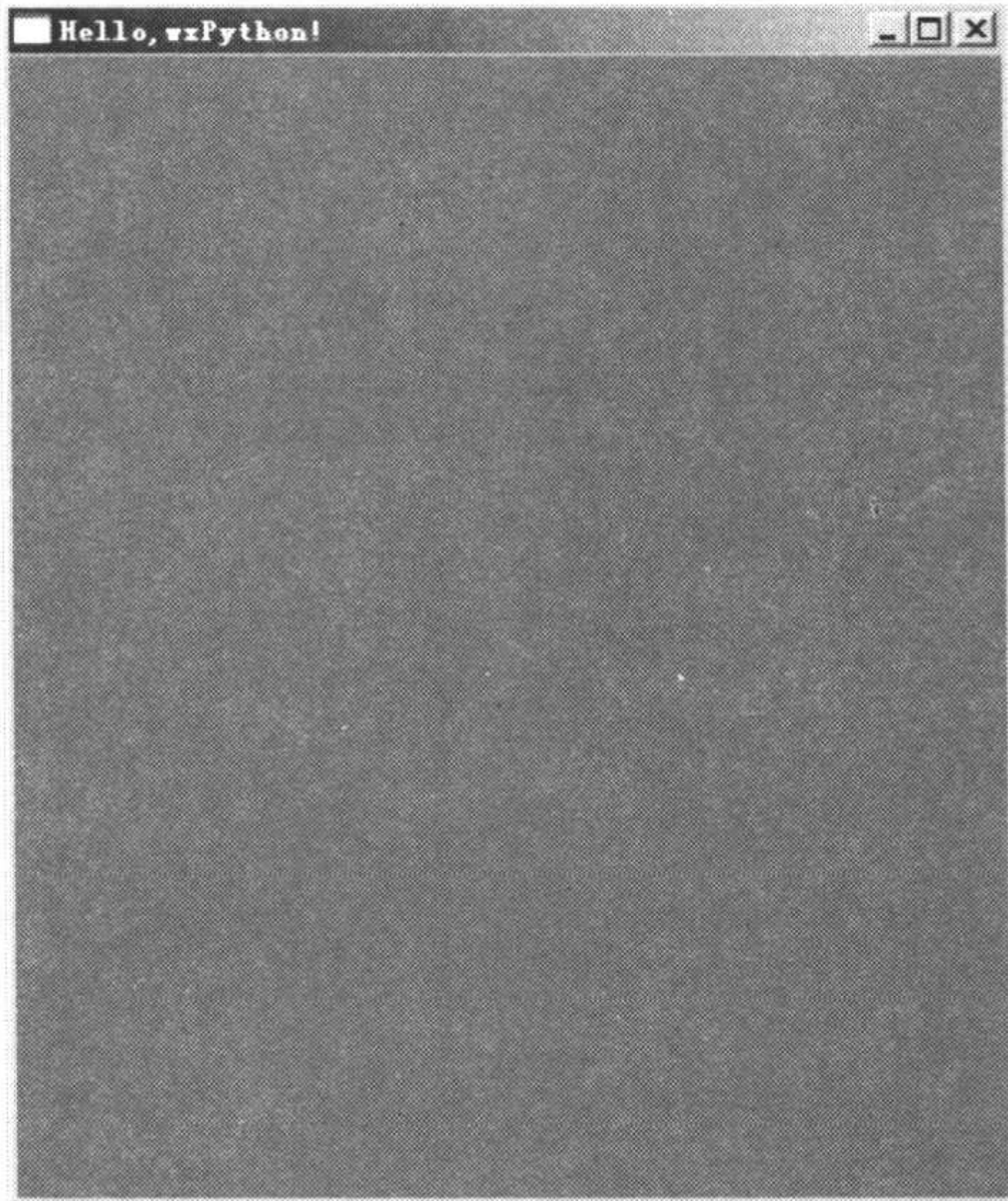


图 13-6 wxPython 窗口

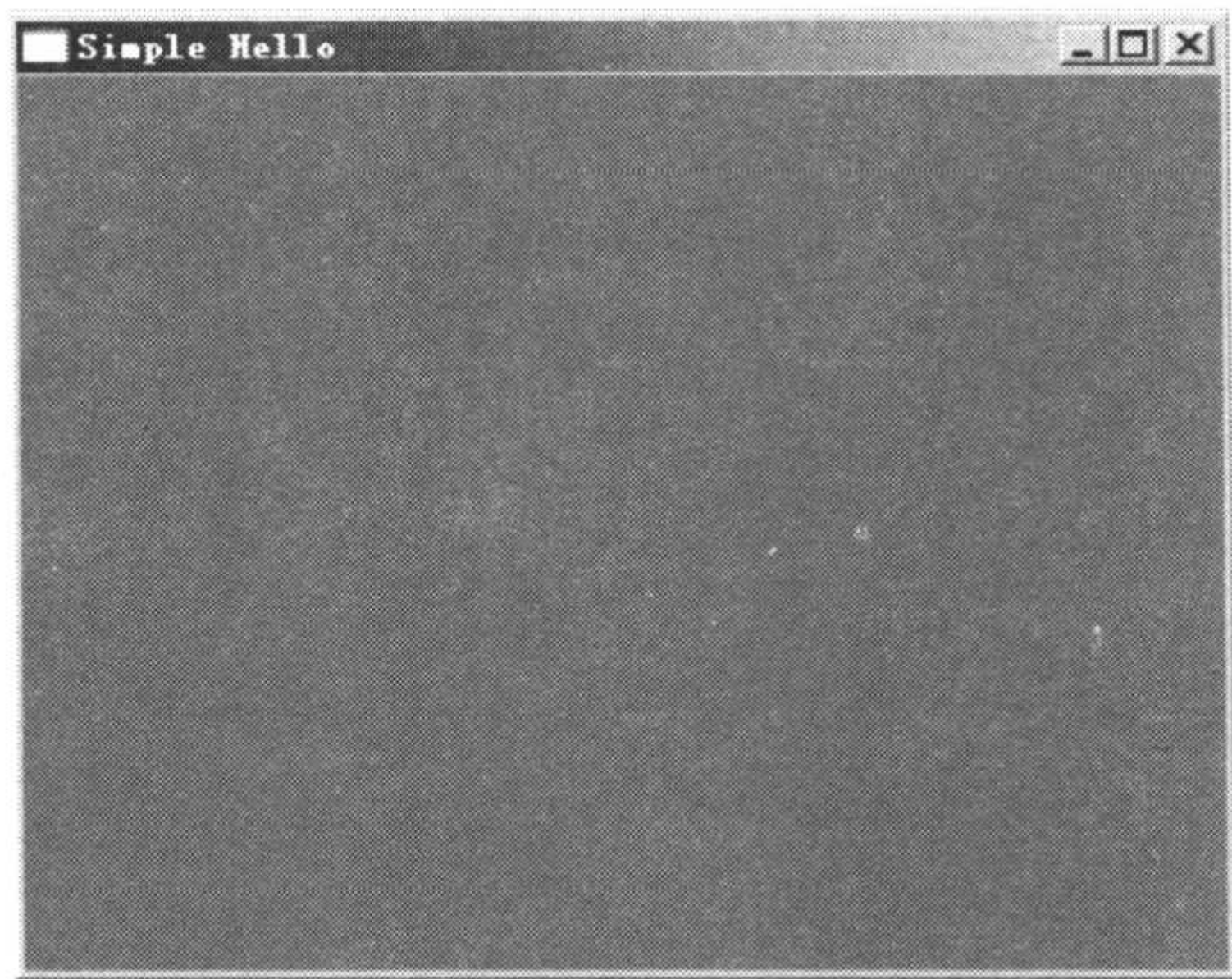


图 13-7 Simple Hello 窗口

在 SimpleHello.py 脚本中，没有继承 wx 的 App 类，而仅使用了 PySimpleApp 生成一个实例对象，然后再创建一个框架窗口，最后进入消息循环。该方法适用于简单窗口的创建，省去了定义类的过程。

13.2 组件

在 wxPython 中只要使用相应的组件类即可创建组件。wxPython 提供了丰富的组件用于完成 GUI 程序的创建。在 wxPython 中所创建的组件应该被包含在框架窗口或者面板中。

13.2.1 面板

在 wxPython 中框架窗口可以容纳其他的组件，但一般情况下框架窗口一般仅包含面板组件，面板组件可以作为“容器”添加其他的组件。使用 wx.Panel 类可以创建一个面板组件。其具有以下初始化参数。除 parent 参数以外，其余的都是可选参数。

- parent: 面板的父组件，一般为所创建的框架窗口。

- id: 面板的 ID, 如果不指定, 可以将其设为-1。
- pos: 面板在父组件中的位置。
- size: 面板的大小。
- style: 面板的样式。
- name: 面板的名字。

如下所示的 wxPythonPanel.py 脚本向框架窗口添加面板。

<pre># -*- coding:utf-8 -*- # file: wxPythonPanel.py # import wx class MyApp(wx.App): def OnInit(self): frame = wx.Frame(parent = None, id=-1, title='Panel', pos=(100,100), size=(600,480), style=wx.DEFAULT_FRAME_STYLE, name="frame") panel = wx.Panel(frame, -1) frame.Show() return True app = MyApp() app.MainLoop()</pre>	<pre># 导入 wxPython 模块 # 通过继承 wx.App 类创建类 # 重载 OnInit 方法 # 创建框架窗口 # 指定框架 ID # 指定窗口标题 # 指定窗口位置 # 指定窗口大小 # 指定窗口样式 # 指定窗口名 # 向框架窗口添加面板 # 显示框架窗口 # 返回 True # 类实例化 # 进入消息循环</pre>
--	---

在 wxPythonPanel.py 脚本中使用了完整的 wx.Frame 初始化参数, 除了 parent 参数以外, 其余都是可以省略的。运行 wxPythonPanel.py 脚本后将创建如图 13-8 所示的窗口。

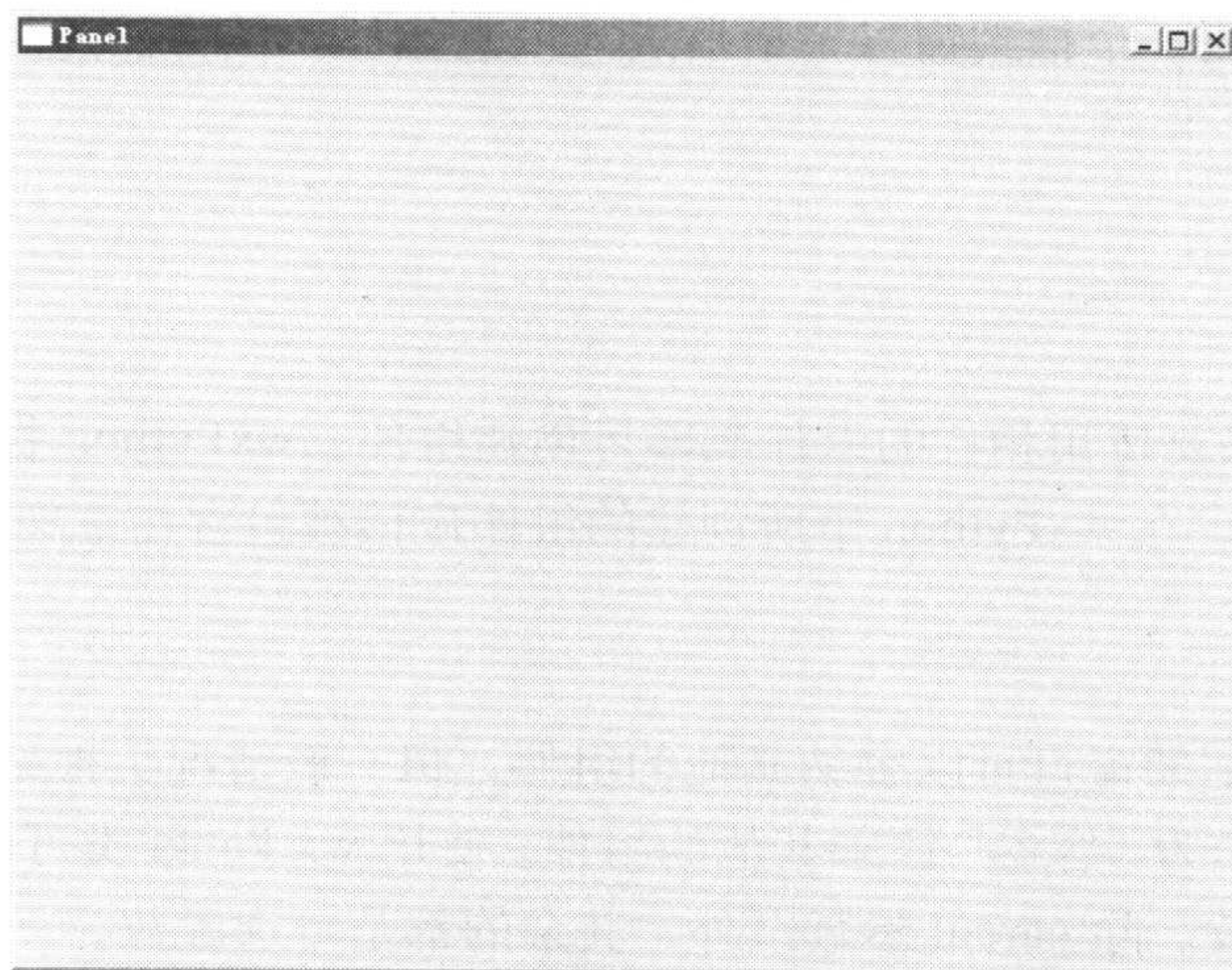


图 13-8 向框架窗口中添加面板

13.2.2 按钮

在 wxPython 中可以使用 wx.Button 类创建按钮。当按钮创建后可以绑定按钮事件，并定义响应按钮事件的函数。

1. 创建按钮

使用 wx.Button 创建按钮时，可以向其传递如下所示的初始化参数。

- parent: 包含该按钮的父组件。
- id: 按钮的 ID。
- label: 按钮所显示的文字。
- pos: 按钮的位置。
- size: 按钮的大小。
- style: 按钮的样式。
- validator: 按钮的验证类。
- name: 按钮名。

如下所示的 wxPythonButton.py 脚本使用 wx.Button 创建按钮。

```
# -*- coding:utf-8 -*-
# file: wxPythonButton.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'Button')
        panel = wx.Panel(frame, -1)
        button = wx.Button(panel,
                           -1,
                           'Button',
                           pos=(50, 50))
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

导入 wxPython
通过继承创建类
重载 OnInit 方法
生成框架窗口
生成面板
向面板添加按钮
指定按钮 ID
指定按钮上的文本
指定按钮在面板上的位置
显示窗口

类实例化
进入消息循环

运行 wxPythonButton.py 脚本后将创建如图 13-9 所示的窗口。

2. 按钮事件处理

绑定按钮事件可以使用 wx.App 类的 Bind 方法，其原型如下所示。

```
Bind(event, handler, source=None, id=-1, id2=-1)
```

其参数含义如下所示。

- event: 所绑定的事件类型。
- handler: 事件的响应函数。

- source: 事件源。
- id: 用于使用组件 ID 代替事件源。
- id2: 用于指定多个组件。

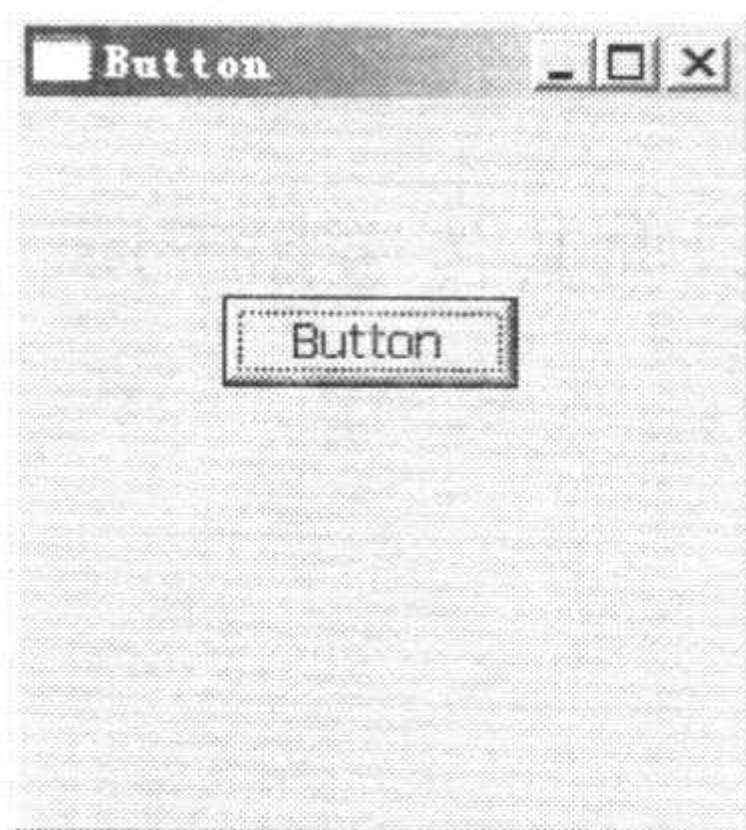


图 13-9 创建按钮

其中事件的响应函数应定义成如下所示的形式。

```
def OnEvent(self, event):
    <处理语句>
```

事件响应函数将接收一个 event 对象的参数，不同的事件，将接收不同的 event 对象。如下所示的 wxPythonButtonEvent.py 使用 wx.App 类的 Bind 方法绑定按钮事件。

```
# -*- coding:utf-8 -*-
# file: wxPythonButtonEvent.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 170))
        panel = wx.Panel(frame, -1)
        self.button1 = wx.Button(panel, -1, 'Button1', pos=(50, 50))
        self.Bind(wx.EVT_BUTTON,
            self.OnButton1,
            self.button1)
        self.button2 = wx.Button(panel, -1, 'Button2', pos = (150, 50))
        self.Bind(wx.EVT_BUTTON,
            self.OnButton2,
            self.button2)
        self.button1.SetDefault()
        frame.Show()
        return True
    def OnButton1(self, event):
        self.button2.SetLabel('Button1')
        self.button2.SetDefault()
        self.button1.SetLabel('Button2')
    def OnButton2(self, event):
        self.button1.SetLabel('Button1')
```

导入 wxPython
通过继承创建类
重载 OnInit 方法
生成框架窗口
生成面板
添加 Button1
绑定按钮事件
指定事件响应函数
指定按钮
绑定按钮事件
指定事件响应函数
指定按钮
将 Button1 设为默认按钮
显示窗口
按钮事件响应函数
更改 Button2 的文字
将 Button2 设为默认按钮
更改 Button1 的文字
按钮事件响应函数
更改 Button1 的文字

第13章 使用 wxPython 编写 GUI

```

        self.button1.SetDefault()
        self.button2.SetLabel('Button2')
app = MyApp()
app.MainLoop()

```

```

# 将 Button1 设为默认按钮
# 更改 Button2 的文字

```

运行 wxPythonButtonEvent.py 脚本，单击【Button1】按钮，将改变按钮的标题，如图 13-10 所示。

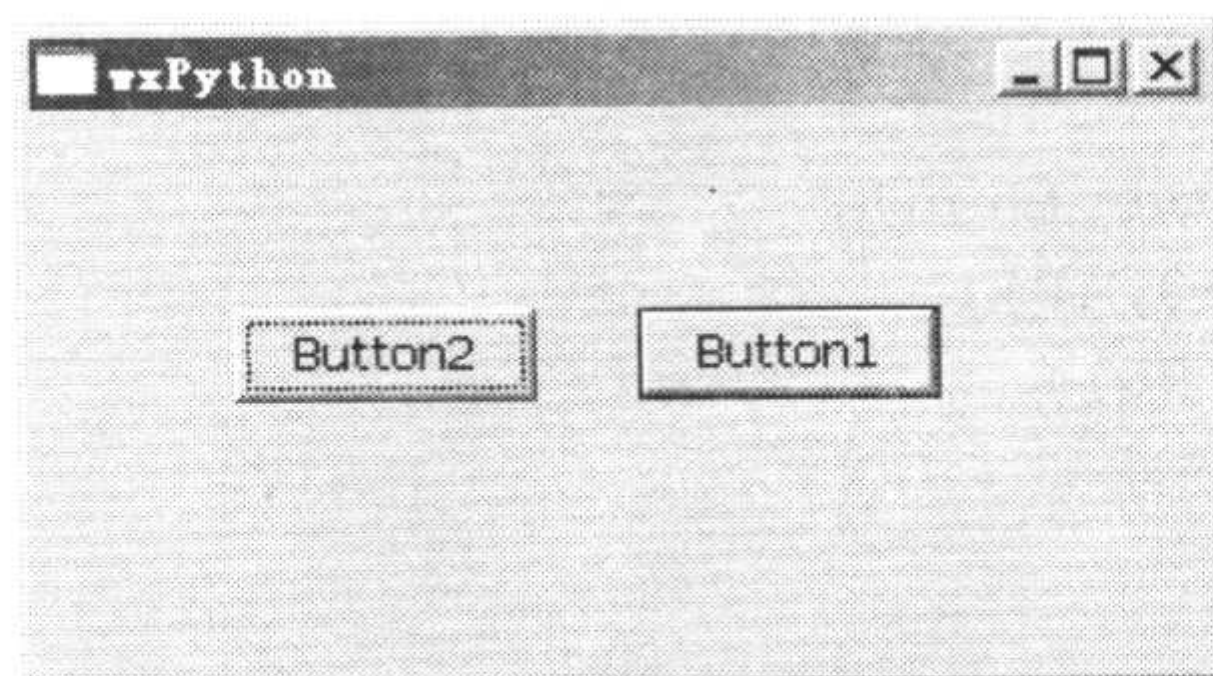


图 13-10 绑定按钮事件

13.2.3 标签

使用 wx.StaticText 类可以创建标签。其初始化参数如下所示。

- parent: 包含该标签的父组件。
- id: 标签的 ID。
- label: 标签所显示的文本。
- pos: 标签的位置。
- size: 标签的大小。
- style: 标签的样式。
- name: 标签名。

如下所示的 wxPythonStatic.py 脚本使用 wx.StaticText 类创建标签。

```

# -*- coding:utf-8 -*-
# file: wxPythonStatic.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 200))
        panel = wx.Panel(frame, -1)
        label1 = wx.StaticText(panel,
                                -1,
                                'Python',
                                size = (160, 20),
                                pos = (60, 10),
                                style = wx.ALIGN_RIGHT)
        label2 = wx.StaticText(panel,

```

```

# 生成框架窗口
# 生成面板
# 生成标签
# 指定标签 ID
# 指定标签中文本
# 指定标签大小
# 指定标签位置
# 指定标签样式，右对齐
# 生成标签

```



```

        -1,
        'Python',
        size = (160,20),
        pos = (60,50),
        style = wx.ALIGN_CENTER)
label2.SetForegroundColour('red')
label2.SetBackgroundColour('black')
label3 = wx.StaticText(panel,
        -1,
        'Python\nwxPython',
        size = (160,40),
        pos = (60,90))
frame.Show()
return True
app = MyApp()
app.MainLoop()

```

指定标签 ID
 # 指定标签中文本
 # 指定标签大小
 # 指定标签位置
 # 指定标签样式，居中对齐
 # 指定标签前景色
 # 指定标签背景色
 # 生成标签
 # 指定标签 ID
 # 在文本中使用换行符
 # 指定标签大小
 # 指定标签位置

运行 wxPythonStatic.py 脚本后将创建如图 13-11 所示的窗口。

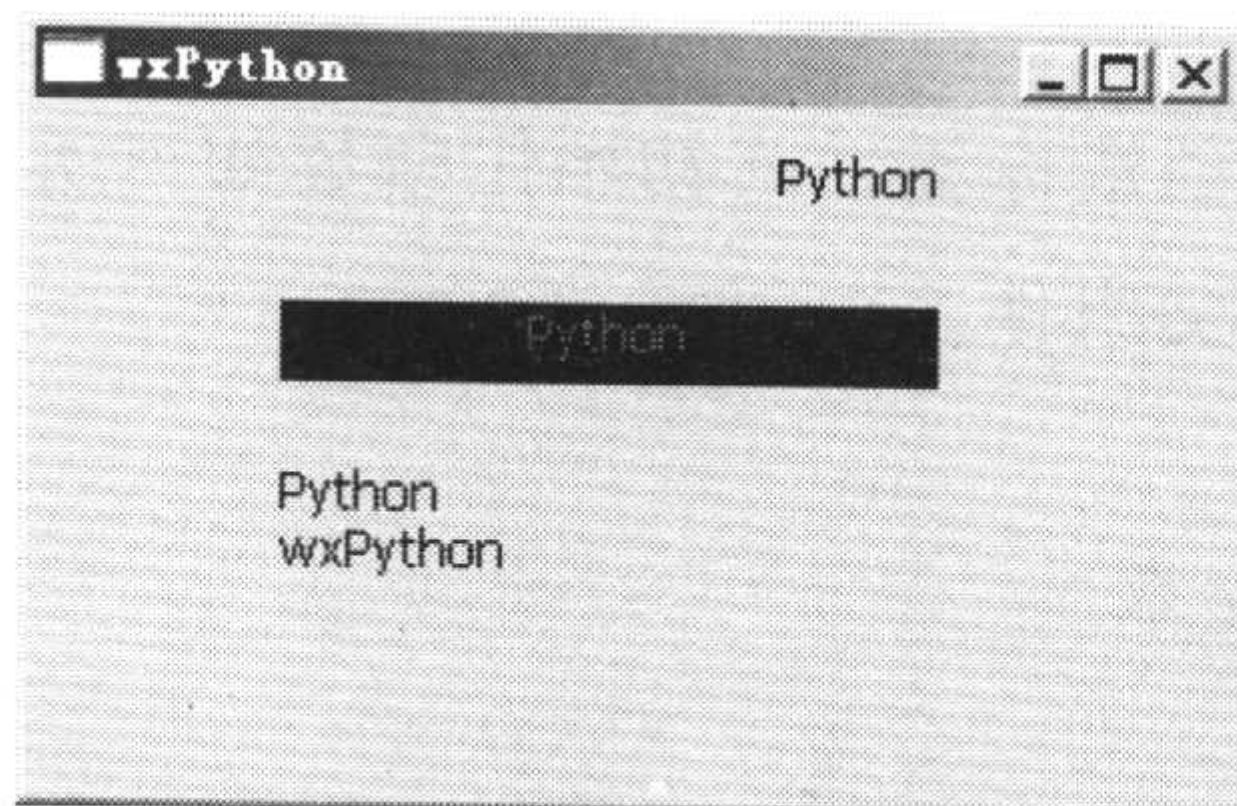


图 13-11 创建标签

13.2.4 文本框

在 wxPython 中使用 wx.TextCtrl 可以创建文本框。通过改变其样式可以创建单行文本框和多行文本框。另外在 wxPython 中还有一类可以设置颜色和字体的文本框。

1. 单行文本框

使用 wx.TextCtrl 创建文本框时，可以向其传递如下所示的参数。

- parent: 包含该文本框的父组件。
- id: 文本框的 ID。
- value: 文本框中的初始文本。
- pos: 文本框的位置。
- size: 文本框的大小。
- style: 文本框的样式。
- validator: 文本框的验证类。
- name: 文本框的名字。

如果不指明文本框的样式，则将创建一个单行文本框。如果指定其样式为 `wx.TE_PASSWORD`，则将创建一个密码框。如下所示的 `wxPythonTextS.py` 脚本创建了一个单行文本框和一个密码框。

```
# -*- coding:utf-8 -*-
# file: wxPythonTextS.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300,200))
        # 生成框架窗口

        panel = wx.Panel(frame, -1)
        # 生成面板

        label1 = wx.StaticText(panel, -1, 'wxPython', pos = (120,20))
        # 生成标签

        label2 = wx.StaticText(panel, -1, 'User Name:', pos = (10,50))
        # 生成标签

        text = wx.TextCtrl(panel,
            -1,
            pos = (100,50),
            size = (160, -1))
        # 生成文本框
        # 指定文本框 ID
        # 指定文本框位置
        # 指定文本框大小

        label3 = wx.StaticText(panel, -1, "Password:", pos = (10,100))
        # 生成标签

        password= wx.TextCtrl(panel,
            -1,
            "password",
            pos = (100,100),
            size = (160, -1),
            style = wx.TE_PASSWORD)
        # 生成文本框
        # 指定文本框 ID
        # 指定初始文本
        # 指定文本框位置
        # 指定文本框大小
        # 指定文本框为密码框

        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

运行 `wxPythonTextS.py` 脚本后将创建如图 13-12 所示的窗口。

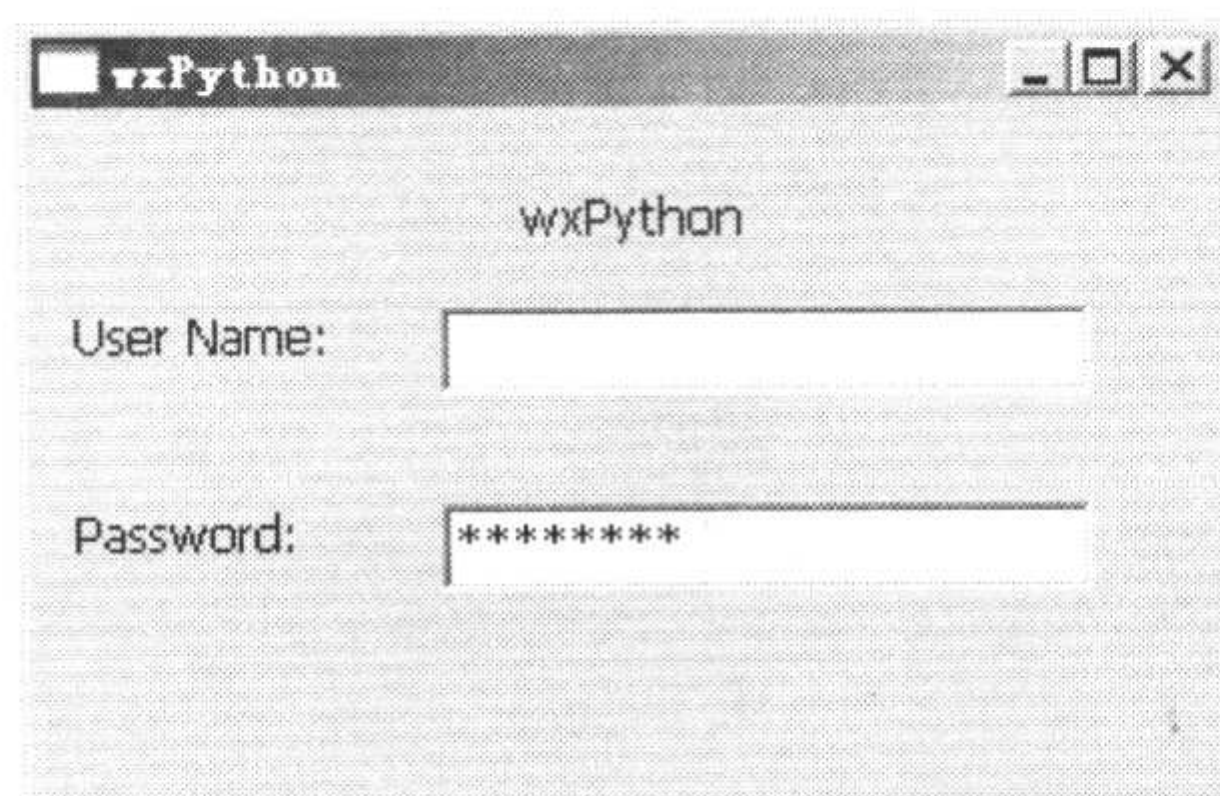


图 13-12 单行文本框和密码框

2. 多行文本框

在创建文本框时，如果指定其样式为 `wx.TE_MULTILINE`，则可以创建一个多行文本框。

如果指定其样式为 `wx.TE_RICH`，则可以创建一个可以改变字体和文本颜色的文本框。如下所示的 `wxPythonTextM.py` 创建了两个多行文本框。

```
# -*- coding:utf-8 -*-
# file: wxPythonTextM.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (600, 400))
        # 生成框架窗口

        panel = wx.Panel(frame, -1)
        # 生成面板

        label1 = wx.StaticText(panel, -1, 'MultiLine', pos = (280, 10))
        # 生成标签

        text1 = wx.TextCtrl(panel,
                             -1,
                             pos = (10, 30),
                             size = (580, 150),
                             style=wx.TE_MULTILINE)
        # 生成文本框
        # 指定文本框 ID
        # 指定文本框位置
        # 指定文本框大小
        # 指定文本框样式

        label2 = wx.StaticText(panel, -1, 'RichText', pos = (280, 190))
        # 生成标签

        text2 = wx.TextCtrl(panel,
                             -1,
                             'Python wxPython',
                             pos = (10, 210),
                             size = (580, 150),
                             style =wx.TE_MULTILINE|wx.TE_RICH)
        # 生成文本框
        # 指定文本框 ID
        # 指定初始文本
        # 指定文本框位置
        # 指定文本框大小
        # 指定文本框样式

        text2.SetStyle(0, 6, wx.TextAttr('red', 'blue'))
        # 指定文本样式

        frame.Show()
        return True

app = MyApp()
app.MainLoop()
```

运行 `wxPythonTextM.py` 脚本后，将创建如图 13-13 所示的窗口。

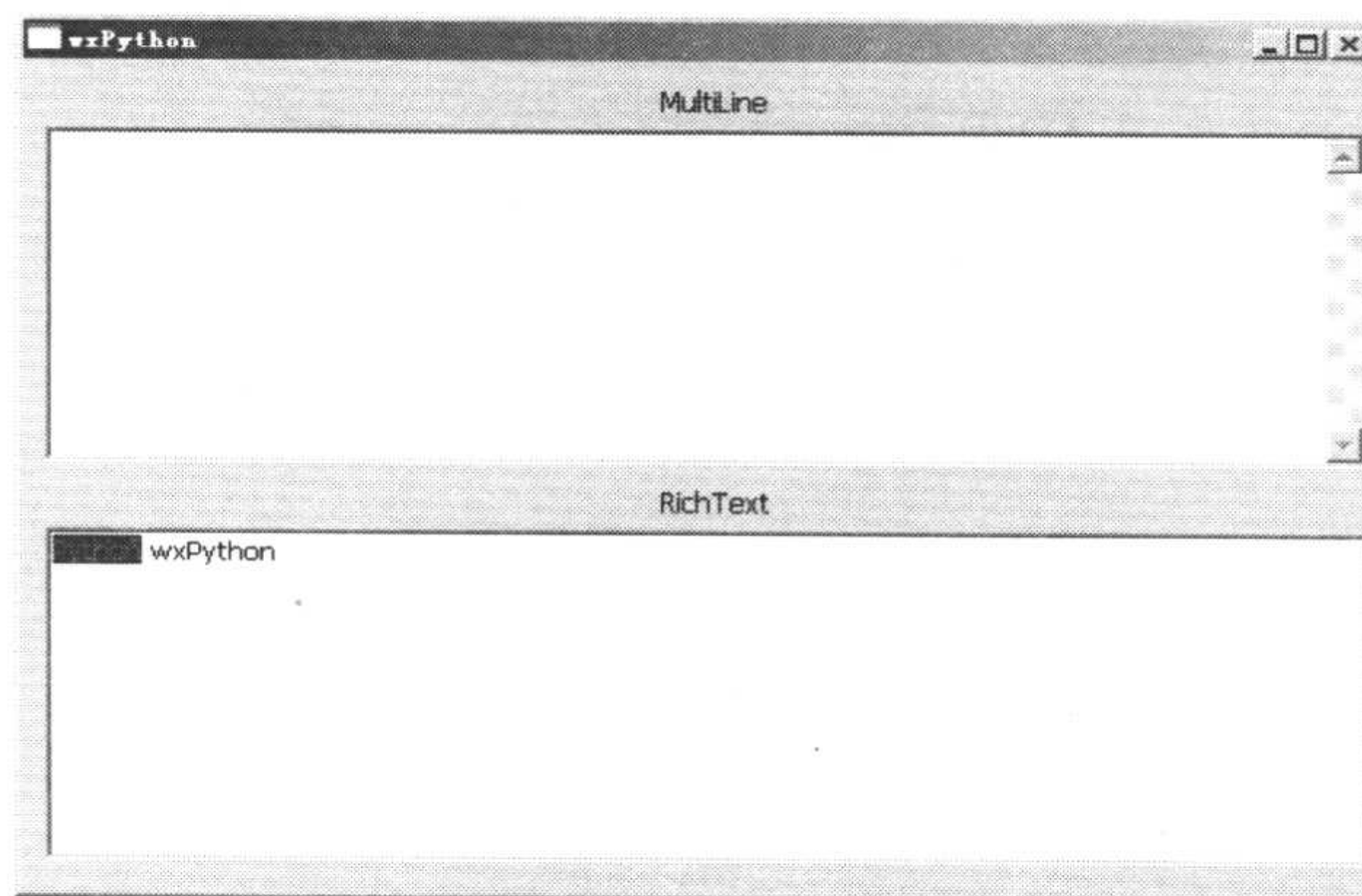


图 13-13 多行文本框

13.2.5 单选框和复选框

在 wxPython 中使用 wx.RadioButton 可以创建单选框，使用 wx.CheckBox 可以创建复选框。使用 wx.RadioButton 创建单选框时可以向其传递如下所示的参数。

- parent: 单选框所在的父组件。
- id: 单选框的 ID。
- label: 单选框所显示的文本。
- pos: 单选框的位置。
- size: 单选框的大小。
- style: 单选框的样式。
- validator: 单选框的验证类。
- name: 单选框的名字。

使用 wx.CheckBox 创建复选框时可以向其传递如下所示的参数。

- parent: 复选框所在的父组件
- id: 复选框的 ID。
- label: 复选框所显示的文本。
- pos: 复选框的位置。
- size: 复选框的大小。
- style: 复选框的样式。
- name: 复选框的名字。

当创建单选框后，可以使用其 GetValue 方法判断单选框是否被选中。如果被选中，GetValue 返回真，否则返回假。同样，使用复选框的 IsChecked 方法可以判断复选框是否被选中。如果被选中，IsChecked 返回真，否则返回假。如下所示 wxPythonCheckRadio.py 创建了单选框和复选框。

```
# -*- coding:utf-8 -*-
# file: wxPythonCheckRadio.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 200))
        panel = wx.Panel(frame, -1)
        self.radiol = wx.RadioButton(panel,
                                     -1,
                                     'Radiol',
                                     # 生成框架窗口
                                     # 生成面板
                                     # 生成单选框
                                     # 指定单选框 ID
                                     # 指定单选框文本
```



```

        pos=(10, 40),
        style=wx.RB_GROUP)
self.radio2 = wx.RadioButton(panel,
    -1,
    'Radio2',
    pos=(10, 80))
self.radio3 = wx.RadioButton(panel,
    -1,
    'Radio3',
    pos=(10, 120))
self.check = wx.CheckBox(panel,
    -1,
    'CheckBox',
    pos = (120, 40),
    size = (150, 20))
self.button1 = wx.Button(panel,-1,'Radio',pos = (120,80))
self.button2 = wx.Button(panel,-1,'Check',pos = (120,120))
self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1)
self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
frame.Show()
return True
def OnButton1(self, event):
    if self.radio1.GetValue():
        self.button1.SetLabel('Radio1')
    elif self.radio2.GetValue():
        self.button1.SetLabel('Radio2')
    else:
        self.button1.SetLabel('Radio3')
def OnButton2(self, event):
    if self.check.IsChecked():
        self.button2.SetLabel('Checked')
    else:
        self.button2.SetLabel('UnChecke')
app = MyApp()
app.MainLoop()

```

指定单选框位置
指定单选框样式
生成框架窗口
指定单选框 ID
指定单选框文本
指定单选框位置
生成单选框
指定单选框 ID
指定单选框文本
指定单选框位置
生成复选框
指定复选框 ID
指定复选框文本
指定复选框位置
指定复选框大小
生成按钮
绑定按钮事件
按钮事件处理方法
判断 Radio1 是否被选中
判断 Radio2 是否被选中
按钮事件处理方法
判断 CheckBox 是否被选中

运行 wxPythonCheckRadio.py 脚本后将创建如图 13-14 所示的窗口。

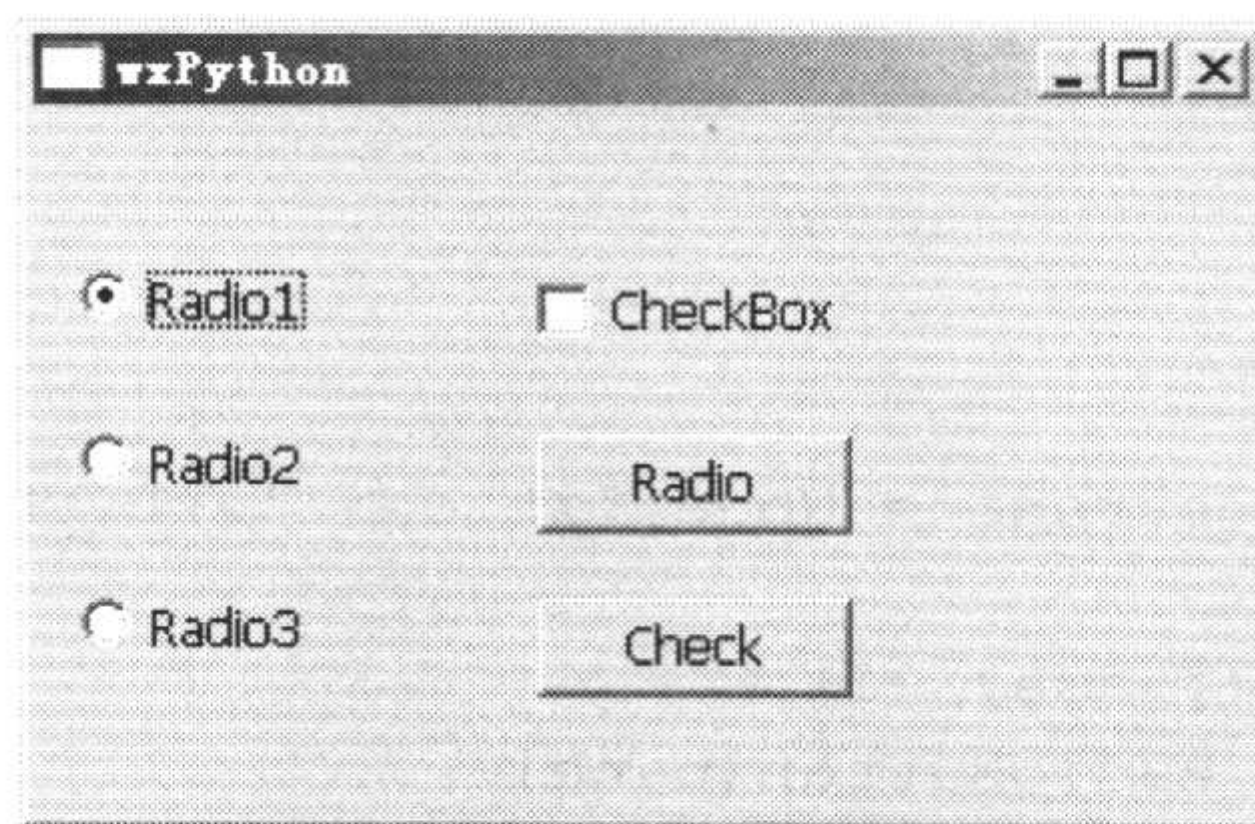


图 13-14 单选框和复选框

13.2.6 使用 sizer 布置组件

在 wxPython 中可以使用 sizer 管理组件的布局。使用 sizer 可以将窗口或其他组件“容器”划分成单元格的形式，然后将组件安排在这些单元格之中，这样就不必设置组件的位置，通过使用 sizer 就可以使组件按所需的形式布置。wxPython 中有 5 种形式的 sizer，如表 13-1 所示。

类 名	描 述
GridSizer	创建基本的网格型 sizer
wx.FlexGridSizer	创建可以根据组件大小改变的 sizer
GridBagSizer	创建可以任意布置组件的 sizer
BoxSizer	创建水平或垂直方向上单行或单列的 sizer
StaticBoxSizer	创建带有边框和标题的 sizer

如下所示的 wxPythonSizer.py 脚本以网格型的 sizer 为例，在面板中布置组件。

```
# -*- coding:utf-8 -*-
# file: wxPythonSizer.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None,title = 'wxPython',size = (300,200))
        panel = wx.Panel(frame, -1)
        sizer = wx.GridSizer(rows=3, cols=3)
        sizer.AddSpacer(0)
        label = wx.StaticText(panel, -1, 'label')
        sizer.Add(label,flag = wx.ALIGN_CENTER)
        sizer.AddSpacer(0)
        button1 = wx.Button(panel, -1, 'Button1')
        sizer.Add(button1,flag = wx.ALIGN_CENTER)
        sizer.AddSpacer(0)
        button2 = wx.Button(panel, -1, 'Button2')
        sizer.Add(button2,flag = wx.ALIGN_CENTER)
        sizer.AddSpacer(0)
        text = wx.TextCtrl(panel, -1, size = (100,20))
        sizer.Add(text)
        sizer.AddSpacer(0)
        panel.SetSizer(sizer)
        frame.Show()
        return True
app = MyApp()
app.MainLoop()
```

生成框架窗口
生成面板
创建一个三行三列的 sizer
向 sizer 中添加一个空项
生成标签
向 sizer 添加标签居中对齐

生成按钮
向 sizer 中添加按钮

生成按钮
向 sizer 中添加按钮

生成文本框
向 sizer 中添加文本框

向面板中添加 sizer

运行 wxPythonSizer.py 脚本后将创建如图 13-15 所示的窗口。

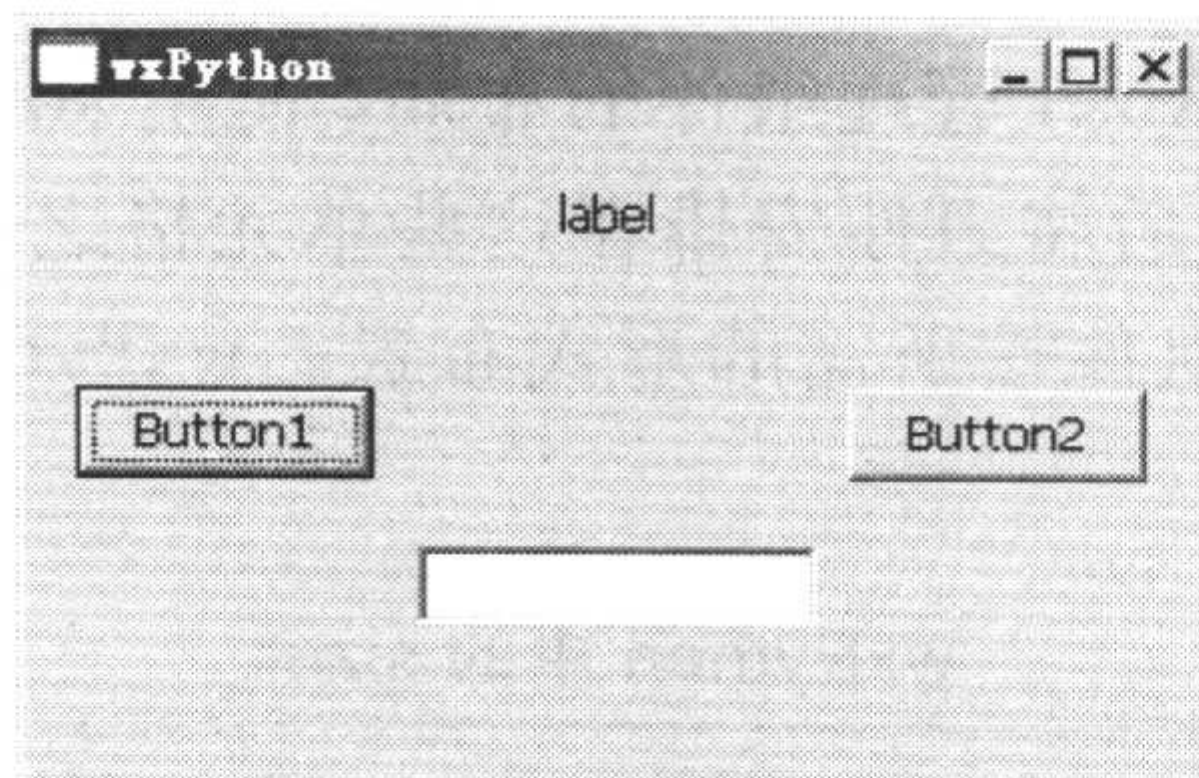


图 13-15 使用 sizer 布置组件

13.3 对话框

wxPython 提供了消息框和标准的对话框。除此以外，wxPython 还提供了对话框类。通过继承 wxPython 的对话框类，可以创建自定义的对话框，实现同用户交互的功能。

13.3.1 消息框和标准对话框

使用 wxPython 中提供的消息框和标准对话框类，可以方便地创建消息框和标准的对话框。其中标准对话框包含了简单的输入框、文件对话框等。

1. 消息框

使用 wx.MessageBox 可以创建消息框。通过组合其样式可以创建不同形式的消息框。其原型如下所示。

```
MessageBox(message, caption, style, parent, x, y)
```

其参数含义如下所示。

- message: 消息框所显示的消息。
- caption: 消息框的标题。
- style: 消息框的样式。
- parent: 消息框的父窗口。
- x: 消息框位置的 x 坐标。
- y: 消息框位置的 y 坐标。

如下所示的 wxPythonMessageBox.py 脚本创建了几种不同样式的消息框。

```
# -*- coding:utf-8 -*-  
# file: wxPythonMessageBox.py  
#  
import wx  
class MyApp(wx.App):  
    def OnInit(self):
```


第13章 使用 wxPython 编写 GUI

```

self.frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 200))
# 生成框架窗口

panel = wx.Panel(self.frame, -1)
# 生成面板

self.button1 = wx.Button(panel, -1, 'Style1', pos = (100, 20))
# 生成按钮

self.button2 = wx.Button(panel, -1, 'Style2', pos = (100, 50))
self.button3 = wx.Button(panel, -1, 'Style3', pos = (100, 80))
self.button4 = wx.Button(panel, -1, 'Style4', pos = (100, 110))
self.button5 = wx.Button(panel, -1, 'Style5', pos = (100, 140))

self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1)
# 绑定按钮事件
self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
self.Bind(wx.EVT_BUTTON, self.OnButton3, self.button3)
self.Bind(wx.EVT_BUTTON, self.OnButton4, self.button4)
self.Bind(wx.EVT_BUTTON, self.OnButton5, self.button5)

self.frame.Show()
return True

def OnButton1(self, event):
    wx.MessageBox('Style1', 'wxPython',
        wx.YES_NO | wx.ICON_QUESTION)
# 按钮事件处理方法
# 创建 MessageBox

def OnButton2(self, event):
    wx.MessageBox('Style2', 'wxPython',
        wx.OK | wx.CANCEL | wx.ICON_ERROR)
# 按钮事件处理方法
# 创建 MessageBox

def OnButton3(self, event):
    wx.MessageBox('Style3', 'wxPython',
        wx.OK | wx.CANCEL | wx.ICON_EXCLAMATION)
# 按钮事件处理方法
# 创建 MessageBox

def OnButton4(self, event):
    wx.MessageBox('Style4', 'wxPython',
        wx.YES_NO | wx.NO_DEFAULT | wx.ICON_HAND)
# 按钮事件处理方法
# 创建 MessageBox

def OnButton5(self, event):
    wx.MessageBox('Style5', 'wxPython',
        wx.YES_NO | wx.YES_DEFAULT | wx.ICON_INFORMATION)
# 按钮事件处理方法
# 创建 MessageBox

app = MyApp()
app.MainLoop()

```

运行 wxPythonMessageBox.py 脚本后, 单击窗口中的按钮将创建如图 13-16~图 13-20 所示的几种消息框。

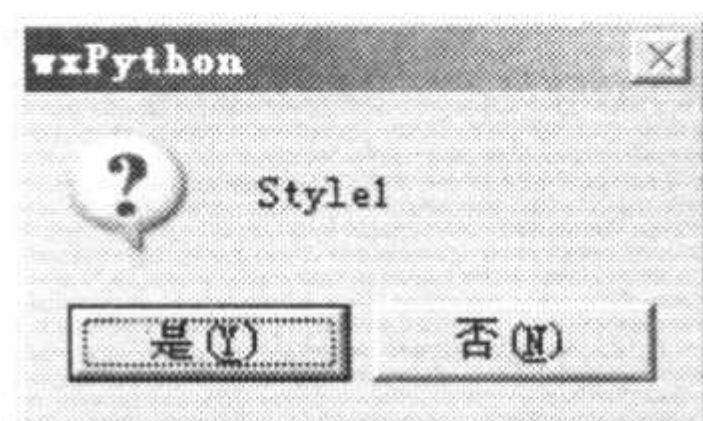


图 13-16 Style1 消息框

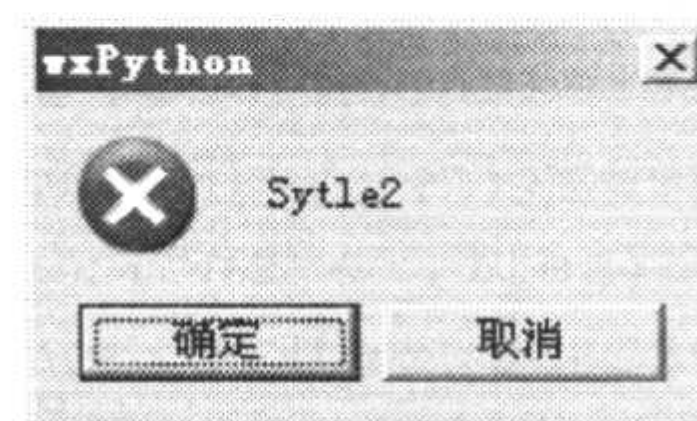


图 13-17 Style2 消息框

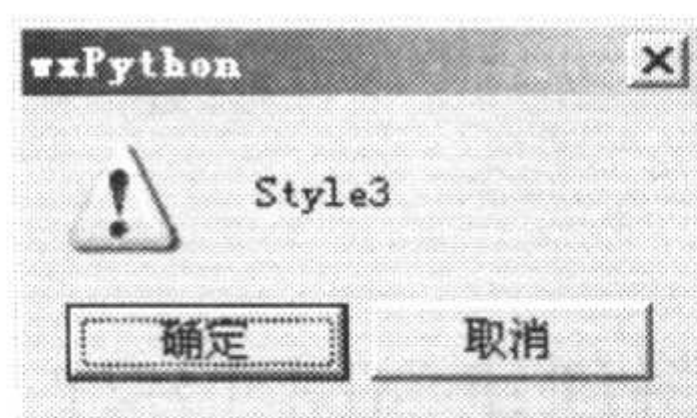


图 13-18 Style3 消息框



图 13-19 Style4 消息框

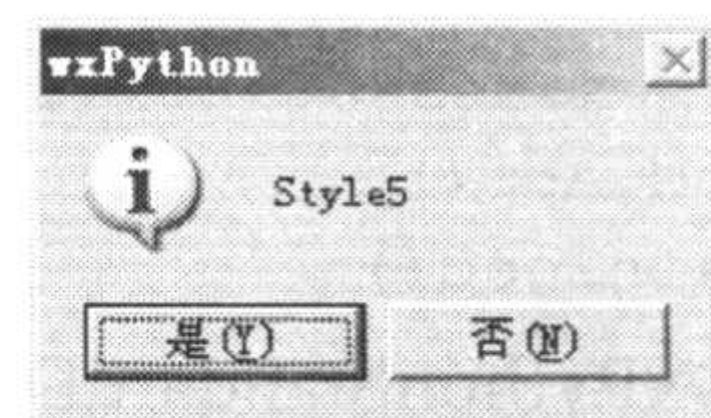


图 13-20 Style5 消息框

2. 标准对话框

wxPython 提供了基本输入对话框以及一些其他的标准对话框，其类型有如下所示的几种。

- wx.GetTextFromUser(): 创建文本输入对话框。
- wx.GetPasswordFromUser(): 创建密码输入对话框。
- wx.GetNumberFromUser(): 创建整数输入对话框。
- wx.FileDialog(): 创建文件打开、关闭对话框。
- wx.FontDialog(): 创建字体选择对话框。
- wx.ColourDialog(): 创建颜色选择对话框。

其中，所创建的标准对话框是 Windows 下标准的对话框样式，与上一章使用 Tkinter 所创建的标准对话框一样。如下所示的 wxPythonStandardDialo.py 仅创建了基本输入对话框。

```
# -*- coding:utf-8 -*-
# file: wxPythonStandardDialo.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        self.frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 200))
        # 生成框架窗口
        panel = wx.Panel(self.frame, -1)
        # 生成面板
        self.button1 = wx.Button(panel, -1, 'Input String', pos = (100, 20))
        # 生成按钮
        self.button2 = wx.Button(panel, -1, 'Input Password', pos = (100, 70))
        self.button3 = wx.Button(panel, -1, 'Input Number', pos = (100, 120))
        self.Bind(wx.EVT_BUTTON, self.OnButton1, self.button1) # 绑定按钮事件
        self.Bind(wx.EVT_BUTTON, self.OnButton2, self.button2)
        self.Bind(wx.EVT_BUTTON, self.OnButton3, self.button3)
        self.frame.Show()
        return True
    def OnButton1(self, event):
        r = wx.GetTextFromUser('wxPython', 'String', 'Default') # 创建文本输入框
        wx.MessageBox(r, 'wxPython', wx.OK) # 创建 MessageBox
    def OnButton2(self, event):
        r = wx.GetPasswordFromUser('wxPython', 'Password') # 创建密码输入框
        wx.MessageBox(r, 'wxPython', wx.OK) # 创建 MessageBox
    def OnButton3(self, event):
        r = wx.GetNumberFromUser('Input Number', 'Number', 'wxPython', 80) # 创建整数输入框
        wx.MessageBox(str(r), 'wxPython', wx.OK) # 创建 MessageBox
app = MyApp()
app.MainLoop()
```

运行 wxPythonStandardDialo.py 脚本后，单击窗口中的按钮，将创建如图 13-21～图 13-23 所示的基本输入对话框。

第13章 使用 wxPython 编写 GUI

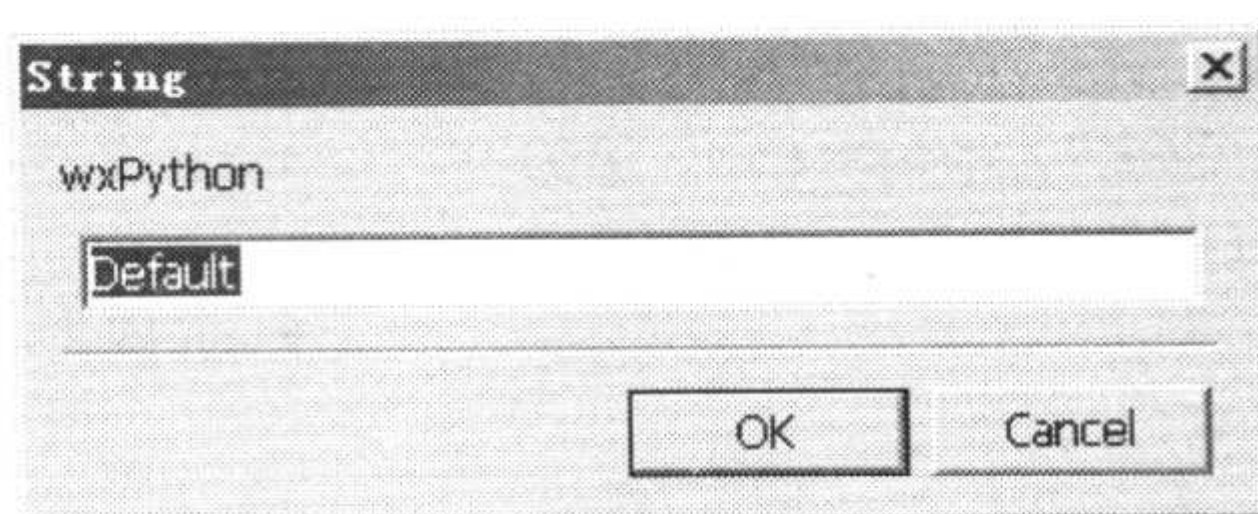


图 13-21 文本输入对话框

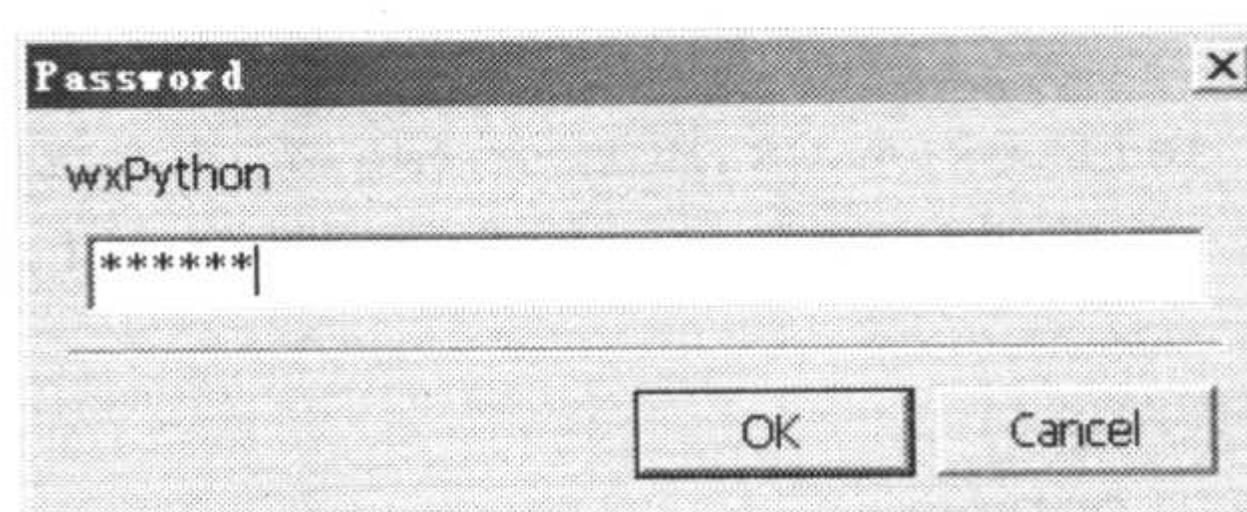


图 13-22 密码输入对话框

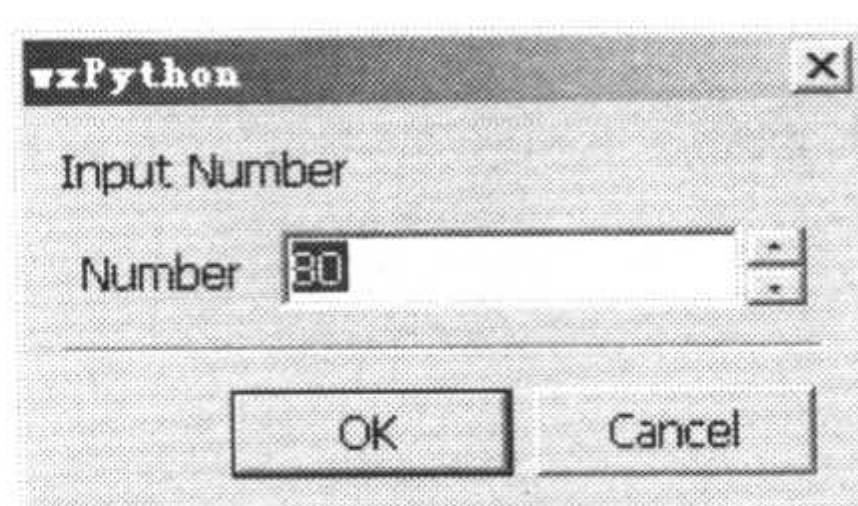


图 13-23 整数输入对话框

13.3.2 创建自定义对话框

wxPython 提供了 wx.Dialog 对话框类，通过继承 wx.Dialog 可以创建自定义的对话框。所创建的对话框可以像创建的框架窗口一样向其中添加其他组件，并绑定组件事件。在创建自己的对话框类时需要在初始化方法中调用 wx.Dialog 的初始化方法。如下所示的 wxPythonDialog.py 脚本使用 wx.Dialog 创建自定义对话框。

```
# -*- coding:utf-8 -*-
# file: wxPythonDialog.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 170))
        panel = wx.Panel(frame, -1)
        self.button = wx.Button(panel, -1, 'Show Dialog', pos=(100, 50))
        self.Bind(wx.EVT_BUTTON, self.OnButton, self.button)
        frame.Show()
        return True
    def OnButton(self, event):
        dialog = MyDialog()
        r = dialog.ShowModal()
        if r == wx.ID_OK:
            wx.MessageBox('You input:' + dialog.text.GetValue(),
                           'wxPython', wx.OK)
            dialog.Destroy()
class MyDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, 'wxDialog', size=(300, 170))
        label = wx.StaticText(self, -1, 'Simple Dialog', pos = (120, 20))
        self.text = wx.TextCtrl(self, -1, pos = (100, 50), size = (160, -1))
```

导入 wxPython
通过继承创建类
重载 OnInit 方法
生成框架窗口
生成面板
添加 Button
显示窗口
按钮事件响应函数
获取返回值
判断是否单击 OK 按钮
弹出消息框
销毁对话框
定义对话框类
初始化
调用父类初始化方法
生成标签


```

self.ok = wx.Button(self, wx.ID_OK, "OK", pos=(50, 80))    # 生成 OK 按钮
self.cancel = wx.Button(self, wx.ID_CANCEL, "Cancel", pos=(200, 80))    # 生成 Cancel 按钮
app = MyApp()
app.MainLoop()

```

运行 wxPythonDialog.py 脚本后，单击【Create】按钮，将创建如图 13-24 所示的对话框。

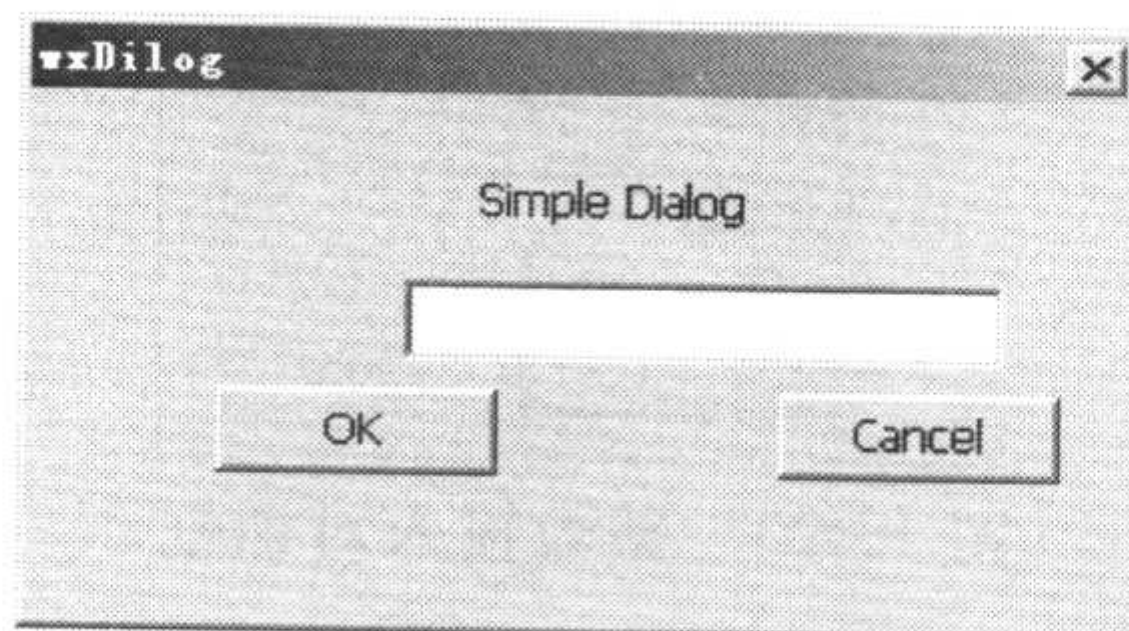


图 13-24 自定义对话框

13.4 菜单

在 wxPython 中创建菜单与使用 PythonWin 中的 MFC 创建菜单过程一样，但要更简单一些。在 wxPython 也可以创建右键菜单等弹出式菜单。

13.4.1 创建菜单

在 wxPython 中创建菜单可以使用 wx.MenuBar 和 wx.Menu。其中，wx.MenuBar 创建的是下拉菜单，而 wx.Menu 则创建的是下拉菜单中的菜单命令。

1. 普通菜单

创建普通菜单时应首先使用 wx.MenuBar 下拉菜单的菜单条。然后使用 wx.Menu 创建菜单命令，当菜单命令创建后，可以调用其 Append 方法添加命令。最后调用下拉菜单的 Append 方法将菜单命令添加到下拉菜单中。如下所示的 wxPythonMenu.py 创建了一组菜单。

```

# -*- coding:utf-8 -*-
# file: wxPythonMenu.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 170))

        panel = wx.Panel(frame, -1)
        menuBar = wx.MenuBar()
        menu = wx.Menu()
        open = menu.Append(-1, 'Open')
        exit = menu.Append(-1, 'Save')
        menu.AppendSeparator()

# 导入 wxPython
# 通过继承创建类
# 重载 OnInit 方法
# 生成框架窗口
# 生成面板
# 创建菜单条
# 创建菜单
# 向菜单中添加 Open
# 向菜单中添加 Save
# 向菜单中添加分隔符

```


第13章 使用 wxPython 编写 GUI

```

close = menu.Append(-1, 'Close')
menuBar.Append(menu, '&File')
menu = wx.Menu()
copy = menu.Append(-1, 'Copy')
paste = menu.Append(-1, 'Paste')
cut = menu.Append(-1, 'Cut')
menu.AppendSeparator()
selectall = menu.Append(-1, 'SelectAll')
menuBar.Append(menu, '&Edit')
menu = wx.Menu()
about = menu.Append(-1, 'About')
menuBar.Append(menu, '&Help')
frame.SetMenuBar(menuBar)
frame.Show()
return True

app = MyApp()
app.MainLoop()

```

向菜单中添加 Close
 # 向菜单条中添加 File
 # 重新创建菜单
 # 向菜单中添加 Copy
 # 向菜单中添加 Paste
 # 向菜单中添加 Cut
 # 向菜单中添加分隔符
 # 向菜单中添加 SelectAll
 # 向菜单条中添加 Edit
 # 重新创建菜单
 # 向菜单中添加 About
 # 向菜单条中添加 Help
 # 向框架窗口中添加菜单
 # 显示窗口

运行 wxPythonMenu.py 脚本后将创建如图 13-25~图 13-27 所示的菜单。

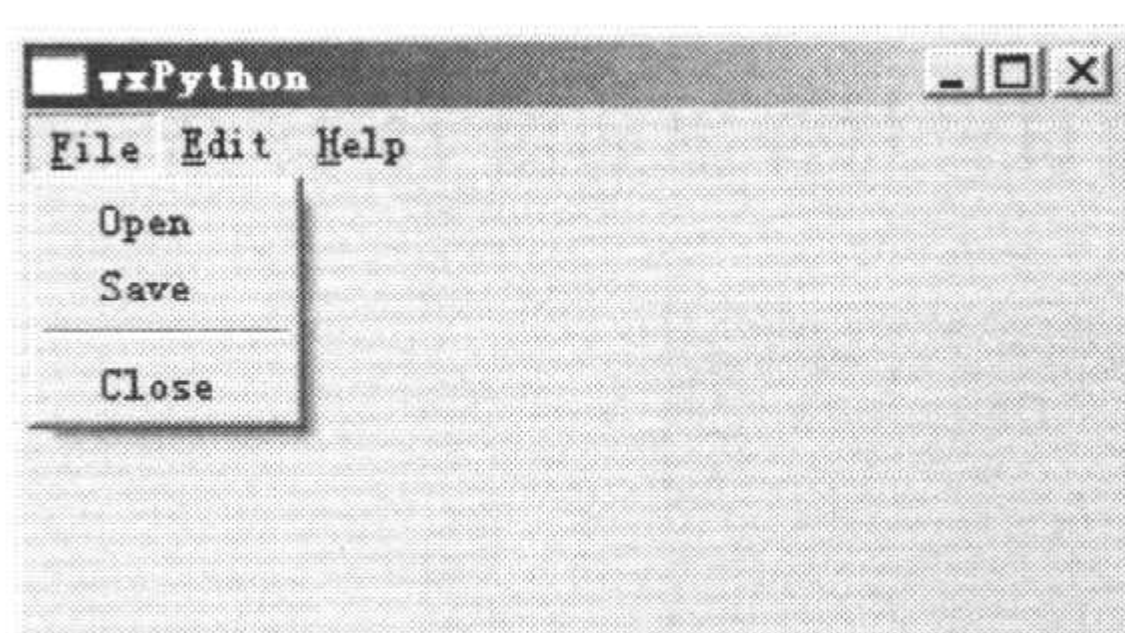


图 13-25 File 菜单

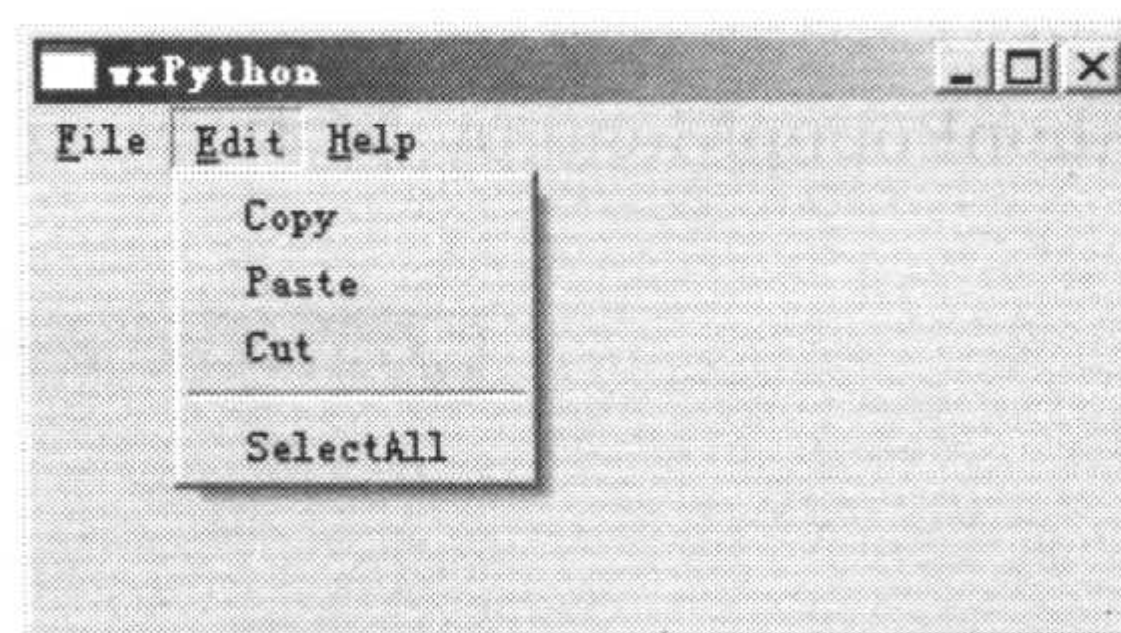


图 13-26 Edit 菜单

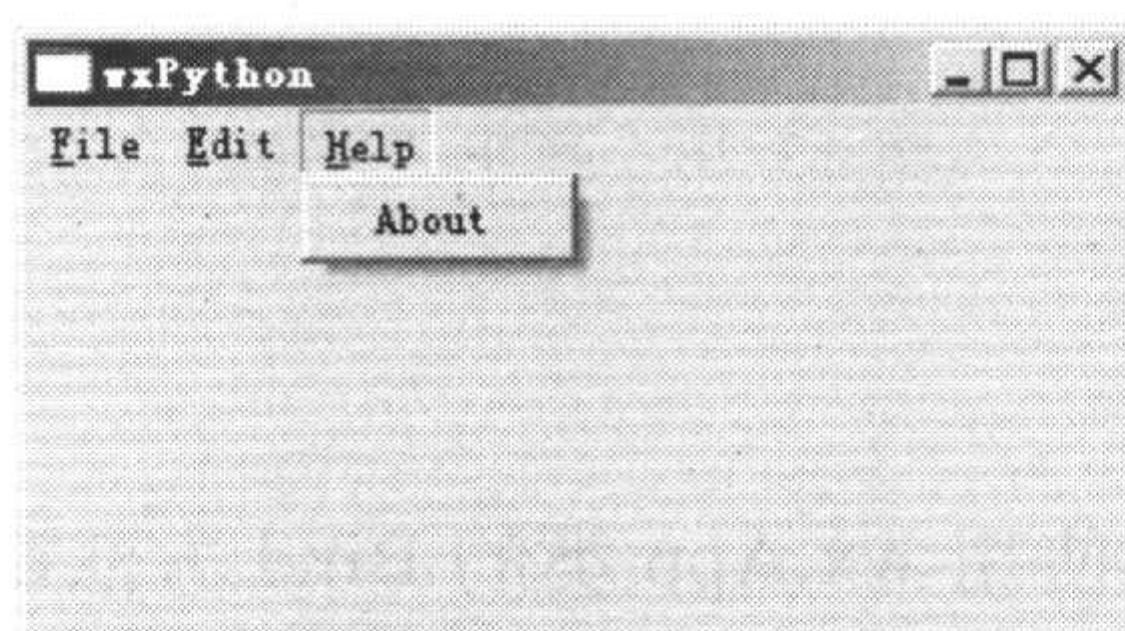


图 13-27 Help 菜单

2. 弹出式菜单

弹出式菜单的创建只需使用 wx.Menu 创建菜单命令，然后使用 PopupMenu 方法显示菜单即可。如下所示的 wxPythonPopupMenu.py 脚本绑定了右击事件，当右击后使用 event 对象的 GetX 和 GetY 方法获取鼠标位置，然后使用 PopupMenu 创建弹出式菜单。

```

# -*- coding:utf-8 -*-
# file: wxPythonPopupMenu.py
#
import wx
class MyApp(wx.App):

```

导入 wxPython
 # 通过继承创建类


```
def OnInit(self):
    frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 170))
    self.panel = wx.Panel(frame, -1)
    menuBar = wx.MenuBar()
    self.menu = wx.Menu()
    open = self.menu.Append(-1, 'Open')
    save = self.menu.Append(-1, 'Save')
    self.menu.AppendSeparator()
    close = self.menu.Append(-1, 'Close')
    menuBar.Append(self.menu, '&File')
    frame.SetMenuBar(menuBar)
    self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
    frame.Show()
    return True

def OnRClick(self, event):
    pos = (event.GetX(), event.GetY())
    self.panel.PopupMenu(self.menu, pos)

app = MyApp()
app.MainLoop()
```

重载 OnInit 方法
生成框架窗口
生成面板
创建菜单条
创建菜单
向框架窗口中添加菜单
绑定右键事件
获得鼠标单击坐标
显示菜单

运行 wxPythonPopupMenu.py 脚本后，右击窗口，将弹出如图 13-28 所示的菜单。

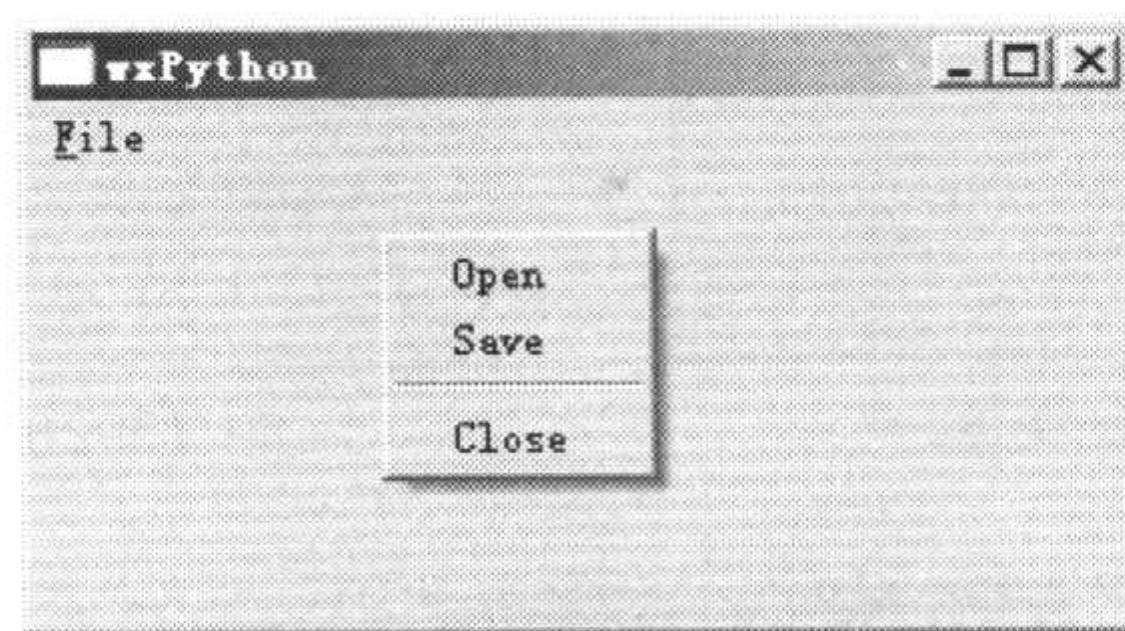


图 13-28 弹出式菜单

13.4.2 绑定菜单事件

同绑定按钮事件一样，只要使用 wx.App 类的 Bind 方法即可。不同的是菜单的绑定事件为 wx.EVT_MENU。如下所示的 wxPythonMenuEvent.py 将菜单绑定了相应的事件处理函数。

```
# -*- coding:utf-8 -*-
# file: wxPythonMenuEvent.py
#
import wx
class MyApp(wx.App):
    def OnInit(self):
        self.frame = wx.Frame(parent = None, title = 'wxPython', size = (300, 170))
        self.panel = wx.Panel(self.frame, -1)
        menuBar = wx.MenuBar()
        self.menu = wx.Menu()
        open = self.menu.Append(-1, 'Open')
```

导入 wxPython
通过继承创建类
重载 OnInit 方法
生成框架窗口
生成面板
创建菜单条
创建菜单

```

        save = self.menu.Append(-1, 'Save')
        self.menu.AppendSeparator()
        close = self.menu.Append(-1, 'Close')
        menuBar.Append(self.menu, '&File')
        self.menu = wx.Menu()
        about = self.menu.Append(-1, 'About')
        menuBar.Append(self.menu, '&Help')
        self.frame.SetMenuBar(menuBar)
        self.Bind(wx.EVT_MENU, self.OnOpen, open)
        self.Bind(wx.EVT_MENU, self.OnSave, save)
        self.Bind(wx.EVT_MENU, self.OnClose, close)
        self.Bind(wx.EVT_MENU, self.OnAbout, about)
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
        self.frame.Show()
        return True
    def OnOpen(self, event):
        dialog = wx.FileDialog(None, 'wxPython', style = wx.OPEN)
        dialog.ShowModal()
        dialog.Destroy()
    def OnSave(self, event):
        dialog = wx.FileDialog(None, 'wxPython', style = wx.SAVE)
        dialog.ShowModal()
        dialog.Destroy()
    def OnClose(self, event):
        self.frame.Destroy()
    def OnAbout(self, event):
        wx.MessageBox('wxPython Menu Event', 'wxPython', wx.OK)
    def OnRClick(self, event):
        pos = (event.GetX(), event.GetY())
        self.panel.PopupMenu(self.menu, pos)
app = MyApp()
app.MainLoop()

```

重新创建菜单

向框架窗口中添加菜单

绑定菜单事件

处理 Open 命令

创建打开文件对话框

处理 Save 命令

创建保存文件对话框

处理 Close 命令

退出程序

处理 About 命令

创建消息框

处理右键事件

创建弹出式菜单

13.5 资源文件

除了直接在脚本中创建组件以外，还可以使用 wxPython 的资源文件定义组件布局。通过 wxPython 的 xrc 模块就可以在脚本中使用资源文件创建 GUI 界面。资源文件可以由专门的软件生成，使用资源文件可以提高脚本的可维护性，减少脚本中的代码。

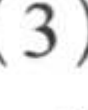
13.5.1 创建资源文件


wxPython 的资源文件使用 XML 的语法，其后缀名为“.xrc”。在“.xrc”文件中每个组件对应于 XML 中的一个 object 节点。其 name 属性为控件的标识，在 Python 脚本中就是通过 name 属性来使用组件的。由于资源文件可以使用 XRCed 等资源编辑软件生成，所以用户不必了解资源文件的细节。


以 wxPython 的文档和例子安装文件中所提供的 XRCed 为例，创建 wxPython 的资源文件过程如下所示。

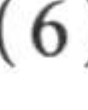
(1) 单击【开始】|【所有应用程序】|【wxPython2.8 Docs Demos and Tools】|【Resource Editor】命令，打开 XRCed，如图 13-29 所示。

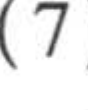
(2) 将【encoding】文本框中的“cp936”改为“utf-8”。

(3) 单击左侧的  按钮，添加一个 frame 组件。将右侧【XML ID】文本框中的内容改为“frame”，【XML ID】即在 Python 脚本中载入组件所使用的资源标识。将【title】文本框中的内容改为“wxPython XRC”。选中【sizer】复选框，在文本框中输入 panel 的大小“300,200”。

(4) 单击左侧的  按钮，添加一个面板组件。将右侧【XML ID】文本框中的内容改为“panel”。

(5) 单击左侧的  按钮，添加一个标签组件。将右侧【XML ID】文本框中的内容改为“label”。在【label】文本框中输入“Input”。选中【pos】复选框，在文本框中输入标签组件的位置“50,70”。

(6) 单击左侧的  按钮，添加一个文本框组件。将右侧【XML ID】文本框中的内容改为“text”。选中【pos】复选框，在文本框中输入标签组件的位置“120,70”。

(7) 单击左侧的  按钮，添加一个按钮组件。将右侧【XML ID】文本框中的内容改为“button”。在【label】文本框中输入“OK”。选中【pos】复选框，在文本框中输入标签组件的位置“100,120”。

(8) 单击工具栏中的  按钮，可以预览创建的资源，如图 13-30 所示。

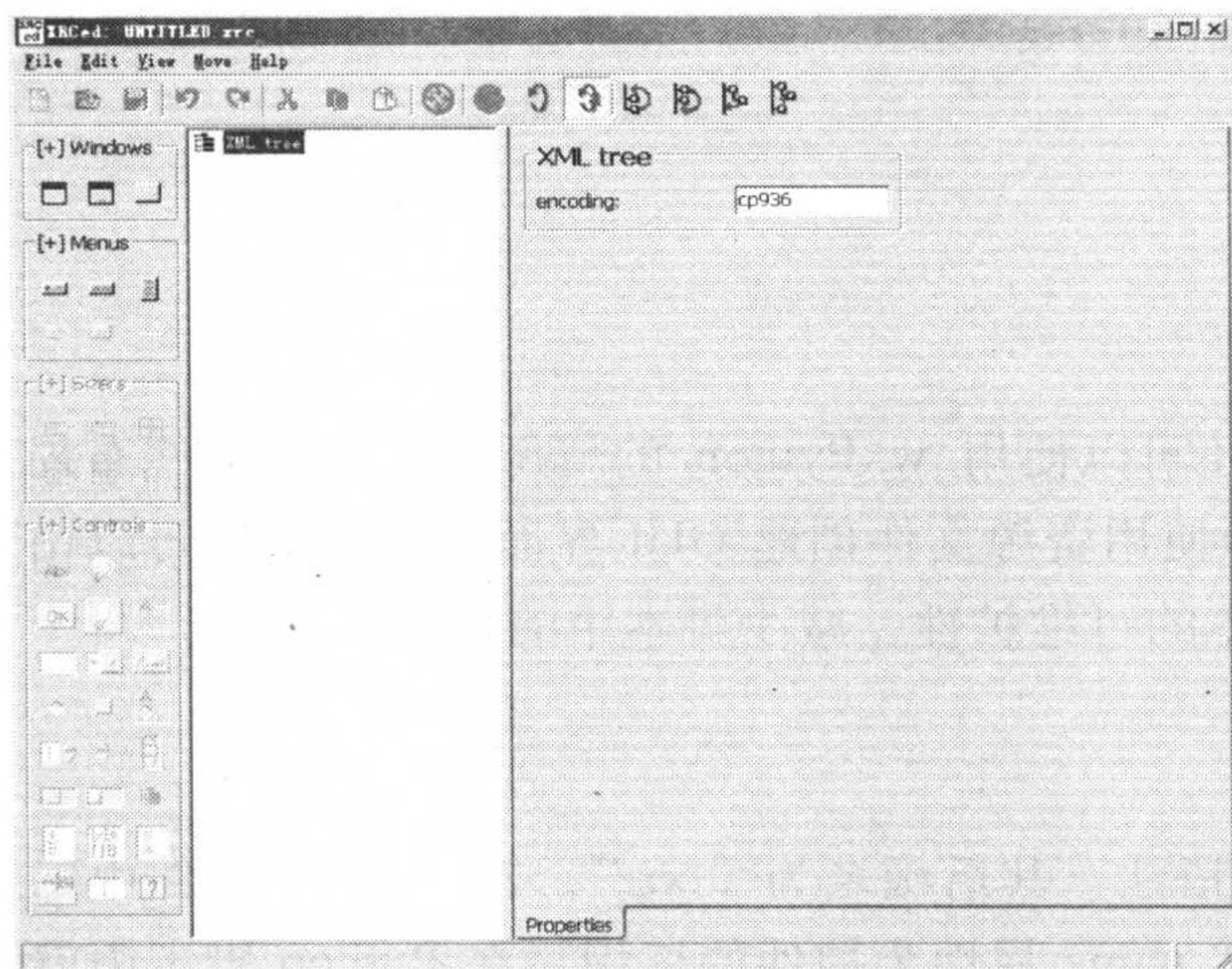


图 13-29 XRCed 资源编辑器

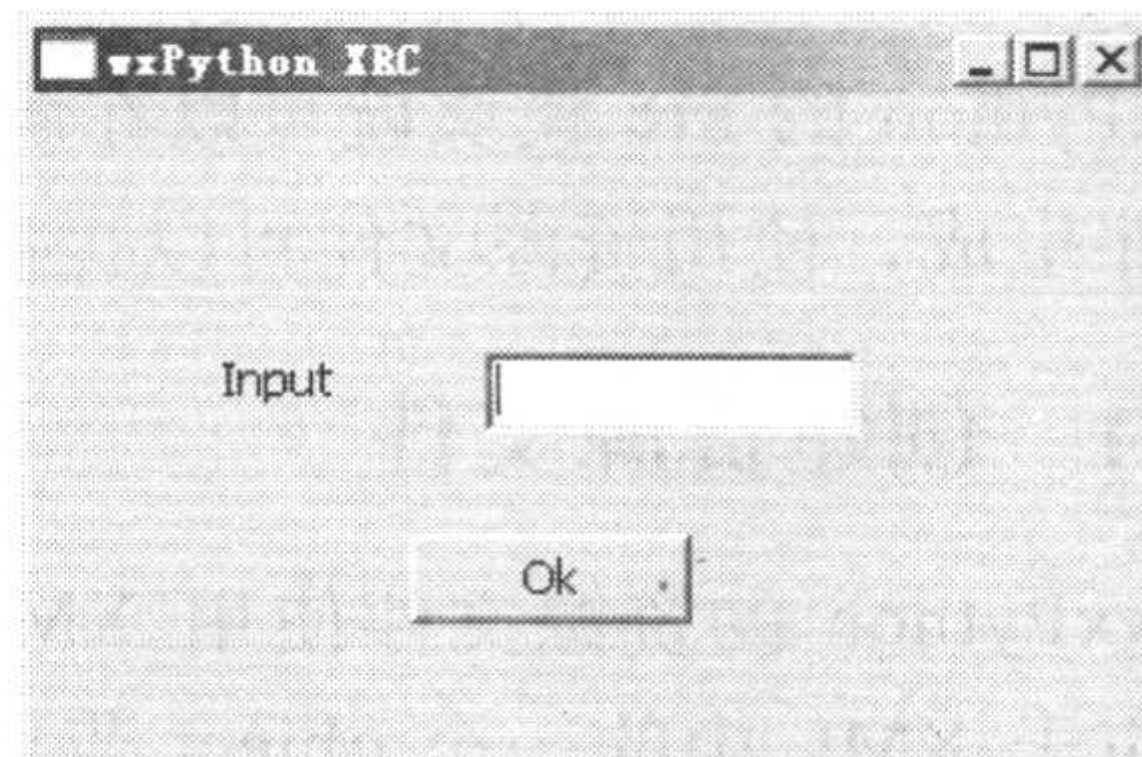


图 13-30 窗口预览

(9) 将资源文件保存为“res.xrc”。

用文本编辑器打开“res.xrc”文件可以看到如下所示的内容。

```
<?xml version="1.0" encoding="cp936"?>
<resource>
  <object class="wxFrame" name="frame">
    <title>wxPython XRC</title>
    <object class="wxPanel" name="panel">
      <object class="wxStaticText" name="label">
        <label>Input</label>
        <pos>50,70</pos>
      </object>
      <object class="wxTextCtrl" name="text">
        <pos>120,70</pos>
      </object>
      <object class="wxButton" name="button">
        <label>Ok</label>
        <pos>100,120</pos>
      </object>
    </object>
    <size>300,200</size>
  </object>
</resource>
```

13.5.2 在脚本中使用资源文件

在脚本中使用资源文件首先要导入 xrc 模块，然后使用 xrc 模块中的 XmlResource 载入资源文件。当资源文件载入以后，首先要使用资源文件对象的 LoadFrame 方法载入框架窗口，然后再使用 xrc 模块中的 XRCCTRL 载入相应的组件。使用 XRCCTRL 需要向其传递两个参数，分别为组件所在的父组件和创建组件，在【XML ID】文本框所填写 ID。

如下所示 wxPythonXRC.py 脚本使用上一小节中创建的“res.xrc”资源文件创建窗口。

```
# -*- coding:utf-8 -*-
# file: wxPythonXRC.py
#
import wx
from wx import xrc
class MyApp(wx.App):
    def OnInit(self):
        self.res = xrc.XmlResource('res.xrc')           # 载入资源文件
        self.frame = self.res.LoadFrame(None, 'frame')  # 从资源文件中载入 frame
        self.panel = xrc.XRCCTRL(self.frame, 'panel')   # 载入 panel
        self.label = xrc.XRCCTRL(self.panel, 'label')   # 载入 label
        self.text = xrc.XRCCTRL(self.panel, 'text')     # 载入 text
        self.button = xrc.XRCCTRL(self.panel, 'button') # 载入 button
        self.Bind(wx.EVT_BUTTON, self.OnButton, self.button) # 绑定按钮事件
        self.frame.Show()
        return True
    def OnButton(self, event):                           # 处理按钮事件
        wx.MessageBox('You input:'+self.text.GetValue(), 'wxPython', wx.OK)
```

```
app = MyApp(False)
app.MainLoop()
```

运行 wxPythonXRC.py 脚本后，在【Input】文本框中输入“Python”，单击【OK】按钮后，如图 13-31 所示。



图 13-31 使用资源创建窗口

13.6 一个简单的文本编辑器

wxPython 所提供的文本框组件功能十分强大，通过使用 wxPython 提供的文本框组件可以创建一个简单的文本编辑器。如下所示的 wxEditor.py 使用 wxPython 所提供的文本框组件实现了基本的打开、关闭文件，粘贴、复制文本等。另外还可以改变文本框的背景色和前景色，以及窗口的透明度等。

```
# -*- coding:utf-8 -*-
# file: wxEditor.py
#
import wx
class CreateMenu():
    def _init_(self, parent):
        self.menuBar = wx.MenuBar()
        self.file = wx.Menu()
        self.open = self.file.Append(-1, '打开')
        self.save = self.file.Append(-1, '保存')
        self.saveas = self.file.Append(-1, '另存为')
        self.file.AppendSeparator()
        self.close = self.file.Append(-1, '退出')
        self.menuBar.Append(self.file, '文件(&F)')
        self.edit = wx.Menu()
        self.undo = self.edit.Append(-1, '撤销')
        self.redo = self.edit.Append(-1, '重做')
        self.edit.AppendSeparator()
        self.cut = self.edit.Append(-1, '剪切')
        self.copy = self.edit.Append(-1, '复制')
        self.paste = self.edit.Append(-1, '粘贴')
        self.edit.AppendSeparator()
        self.selectall = self.edit.Append(-1, '全选')
```

导入 wxPython
创建菜单类
创建菜单条
创建菜单


```

self.menuBar.Append(self.edit, '编辑(&E)')
self.view = wx.Menu()
self.color = self.view.AppendCheckItem(1051, '设为黑色')
self.trans = self.view.Append(-1, '设置透明度')
self.menuBar.Append(self.view, '查看(&V)')
self.help = wx.Menu()
self.about = self.help.Append(-1, '关于')
self.menuBar.Append(self.help, '帮助(&H)')
parent.SetMenuBar(self.menuBar)
class MyApp(wx.App):
    def OnInit(self):
        self.file = ''
        self.width = 600
        self.height = 480
        self.frame = wx.Frame(parent = None, title = 'wxPython Notebook',
                               size = (self.width, self.height))
        self.panel = wx.Panel(self.frame, -1)
        self.menu = CreateMenu(self.frame)
        self.text = wx.TextCtrl(self.panel,
                                 -1,
                                 pos = (2, 2),
                                 size = (self.width-10, self.height-50),
                                 style = wx.HSCROLL | wx.TE_MULTILINE)
        self.Bind(wx.EVT_MENU, self.OnOpen, self.menu.open)
        self.Bind(wx.EVT_MENU, self.OnSave, self.menu.save)
        self.Bind(wx.EVT_MENU, self.OnSaveAs, self.menu.saveas)
        self.Bind(wx.EVT_MENU, self.OnClose, self.menu.close)
        self.Bind(wx.EVT_MENU, self.OnUndo, self.menu.undo)
        self.Bind(wx.EVT_MENU, self.OnRedo, self.menu.redo)
        self.Bind(wx.EVT_MENU, self.OnCut, self.menu.cut)
        self.Bind(wx.EVT_MENU, self.OnCopy, self.menu.copy)
        self.Bind(wx.EVT_MENU, self.OnPaste, self.menu.paste)
        self.Bind(wx.EVT_MENU, self.OnSelectAll, self.menu.selectall)
        self.Bind(wx.EVT_MENU, self.OnColor, self.menu.color)
        self.Bind(wx.EVT_MENU, self.OnTrans, self.menu.trans)
        self.Bind(wx.EVT_MENU, self.OnAbout, self.menu.about)
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
        self.Bind(wx.EVT_SIZE, self.Resize)
        self.frame.Show()
        return True
    def OnOpen(self, event):
        dialog = wx.FileDialog(None, 'wxPython Notebook', style = wx.OPEN)
        if dialog.ShowModal() == wx.ID_OK:
            self.file = dialog.GetPath()
            file = open(self.file)
            self.text.write(file.read())
            file.close()
        dialog.Destroy()
    def OnSave(self, event):

```

向框架窗口中添加菜单
通过继承创建类
重载 OnInit 方法
生成框架窗口
生成面板
生成菜单
生成文本框
绑定事件
处理打开命令
处理保存命令

```

    if self.file == '':
        dialog = wx.FileDialog(None, 'wxPython Notebook', style = wx.SAVE)
        if dialog.ShowModal() == wx.ID_OK:
            self.file = dialog.GetPath()
            self.text.SaveFile(self.file)
            dialog.Destroy()
        else:
            self.text.SaveFile(self.file)
def OnSaveAs(self, event):                                # 处理另存为命令
    dialog = wx.FileDialog(None, 'wxPython Notebook', style = wx.SAVE)
    if dialog.ShowModal() == wx.ID_OK:
        self.file = dialog.GetPath()
        self.text.SaveFile(self.file)
        dialog.Destroy()
def OnClose(self, event):                                # 处理退出命令
    self.frame.Destroy()
def OnAbout(self, event):                                # 处理关于命令
    wx.MessageBox('A simple editor!', 'wxPython Notebook', wx.OK)
def OnRClick(self, event):                                # 处理右键事件
    pos = (event.GetX(), event.GetY())
    self.panel.PopupMenu(self.menu.edit, pos)
def OnUndo(self, event):                                  # 处理撤销命令
    self.text.Undo()
def OnRedo(self, event):                                  # 处理重做命令
    self.text.Redo()
def OnCut(self, event):                                   # 处理剪切命令
    self.text.Cut()
def OnCopy(self, event):                                  # 处理复制命令
    self.text.Copy()
def OnPaste(self, event):                                 # 处理粘贴命令
    self.text.Paste()
def OnSelectAll(self, event):                             # 处理全选命令
    self.text.SelectAll()
def OnColor(self, event):                                 # 处理设为黑色命令
    if self.menu.view.IsChecked(1051):
        self.text.SetBackgroundColour('black')
        self.text.SetForegroundColour('green')
        self.text.Refresh()
    else:
        self.text.SetBackgroundColour('white')
        self.text.SetForegroundColour('black')
        self.text.Refresh()
def OnTrans(self, event):                                  # 处理设置透明度命令
    r = wx.GetNumberFromUser('请选择透明度', '',
                              'wxPython Notebook', 80, min = 30)
    if r != -1:
        self.frame.SetTransparent((r* 255/100))
        self.frame.Refresh()
def Resize(self, event):                                  # 处理窗口改变大小命令

```

第13章 使用 wxPython 编写 GUI

```

newsiz = self.frame.GetSize()
width = newsiz.GetWidth() - 10
height = newsiz.GetHeight() - 50
self.text.SetSize((width,height))
self.text.Refresh()

app = MyApp()
app.MainLoop()

```

运行 wxEditor.py 后, 打开如图 13-32 所示的窗口。单击【查看】|【设为黑色】命令, 可以将背景设为黑色, 文字设为绿色, 如图 13-33 所示。

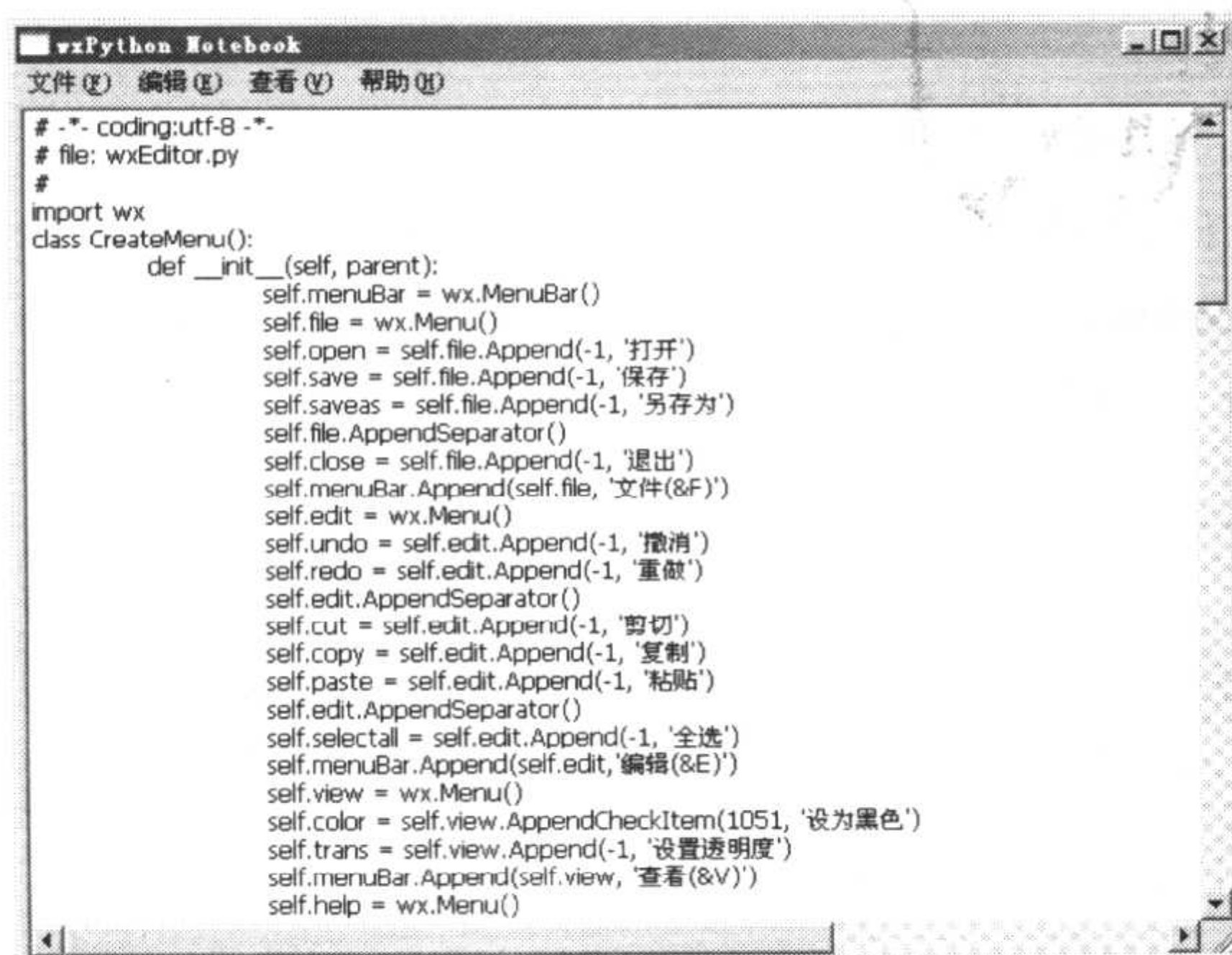


图 13-32 简单的文本编辑器 wxEditor

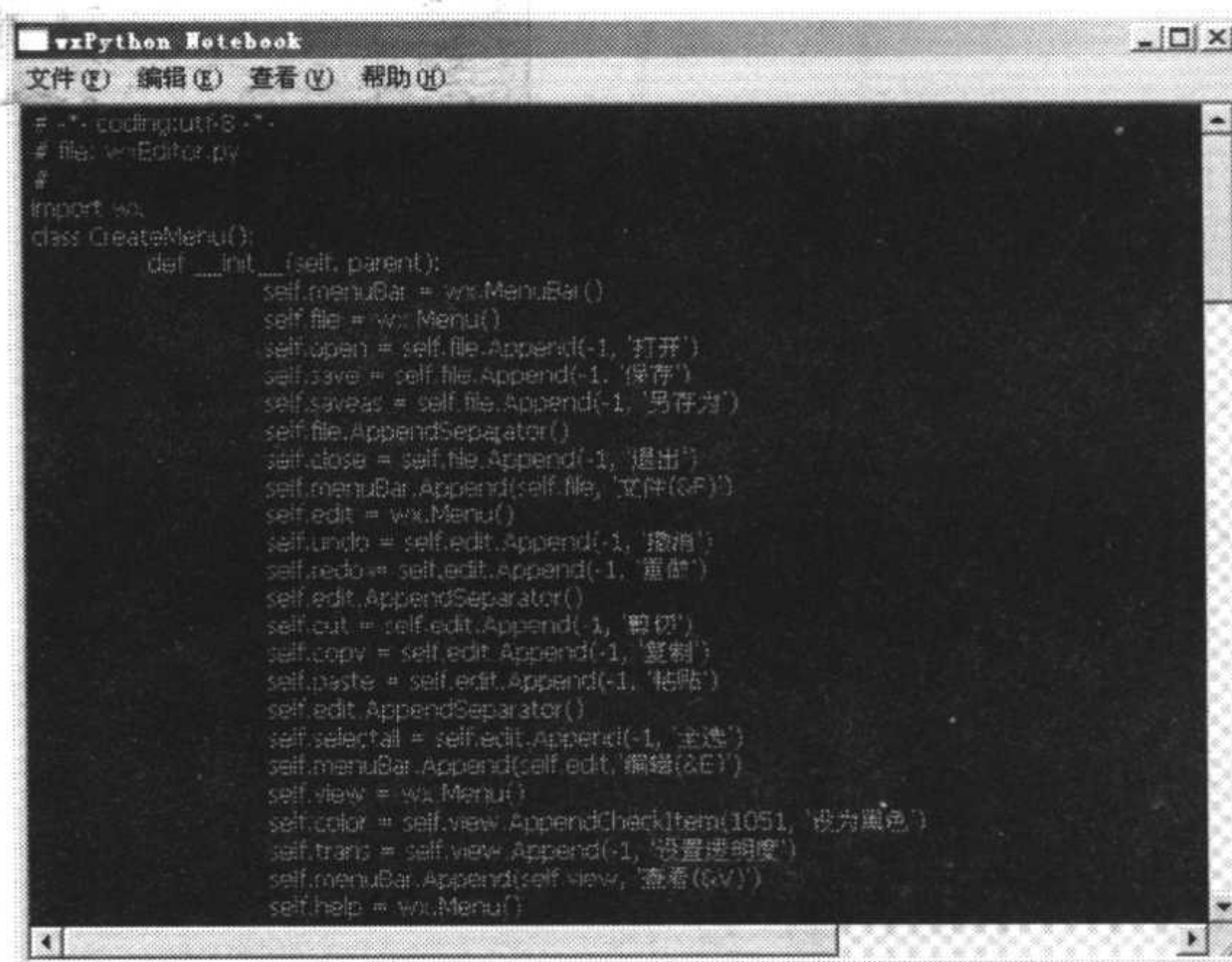


图 13-33 将 wxEditor 背景设为黑色

CHAPTER 14

第 14 章 使用 PyGTK 编写 GUI

PyGTK 是对 GTK 的封装。通过使用 PyGTK 模块可以在 Python 中使用 GTK 来创建 GUI 界面。GTK 是开源的图形用户界面库。虽然 GTK 库是使用 C 语言编写的，但其使用了类的思想。GTK 库可以运行在包括 Windows 在内的多种操作系统上。

14.1 PyGTK 概述

由于 GTK 主要使用在 Linux 平台下，其风格与 Windows 的风格有所不同。在 Windows 下使用 PyGTK 需要使用 GTK 的运行库，PyGTK 官方提供的安装程序包含了 GTK 的运行库。

14.1.1 PyGTK 安装

PyGTK 不是 Python 官方安装程序的一部分，因此使用 PyGTK，要从其官方网站 <http://www.pygtk.org> 下载 PyGTK 的安装程序。在 Windows 下安装 PyGTK，需要安装运行 PyGTK 所需要的动态链接库文件。为了简便起见，推荐下载官方的“PyGTK all-in-one installer for win32”。该安装程序将安装在 Windows 下使用 PyGTK 所需的所有文件，并且设置系统环境变量。PyGTK 的安装步骤如下所示。

- (1) 下载 PyGTK 的安装程序后，直接双击运行，如图 14-1 所示。
- (2) 直接单击【Next】按钮，进入下一步，如图 14-2 所示。此处可以根据需要更改其 PyGTK 运行库的安装目录。
- (3) 当安装完 PyGTK 运行库后，将安装 pygtk 模块和 pycairo 模块，如图 14-3、图 14-4 所示。同其他的 Python 模块一样，安装程序将自动检测 Python 的安装目录。
- (4) 剩下的步骤只需单击【下一步】按钮，按照默认设置安装即可。

由于安装程序设置了系统的环境变量，因此安装完成后需要重新启动系统，使环境变量生效。重启系统后可以在 Python 交互式 shell 中输入如下语句，以验证 PyGTK 是否安装成功。

```
import pygtk
import gtk
```

如果上述语句成功运行，则表示 PyGTK 已经安装成功，可以使用 PyGTK 进行 GUI 编程了。

第14章 使用PyGTK 编写GUI

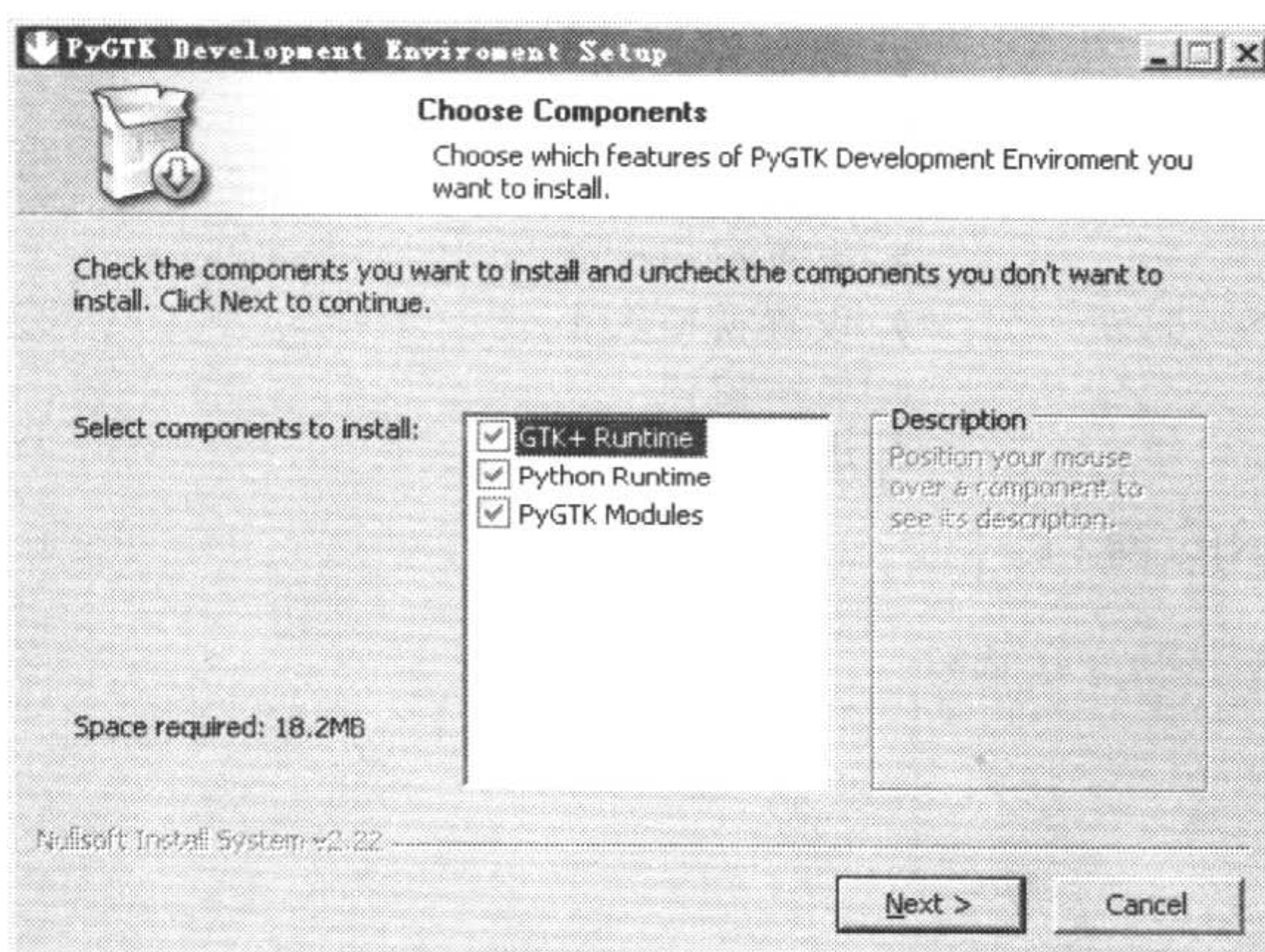


图 14-1 PyGTK 安装程序

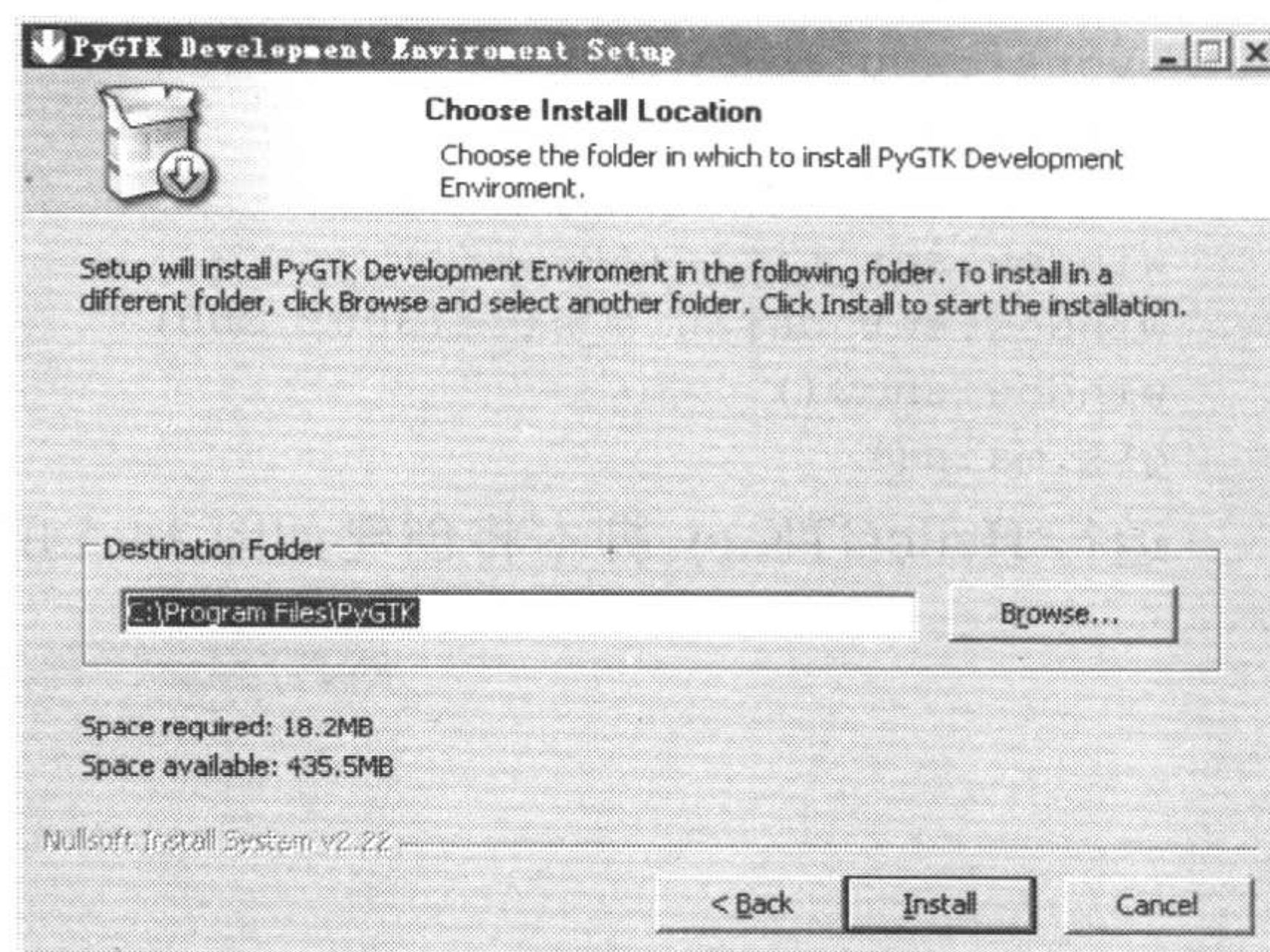


图 14-2 选择安装目录

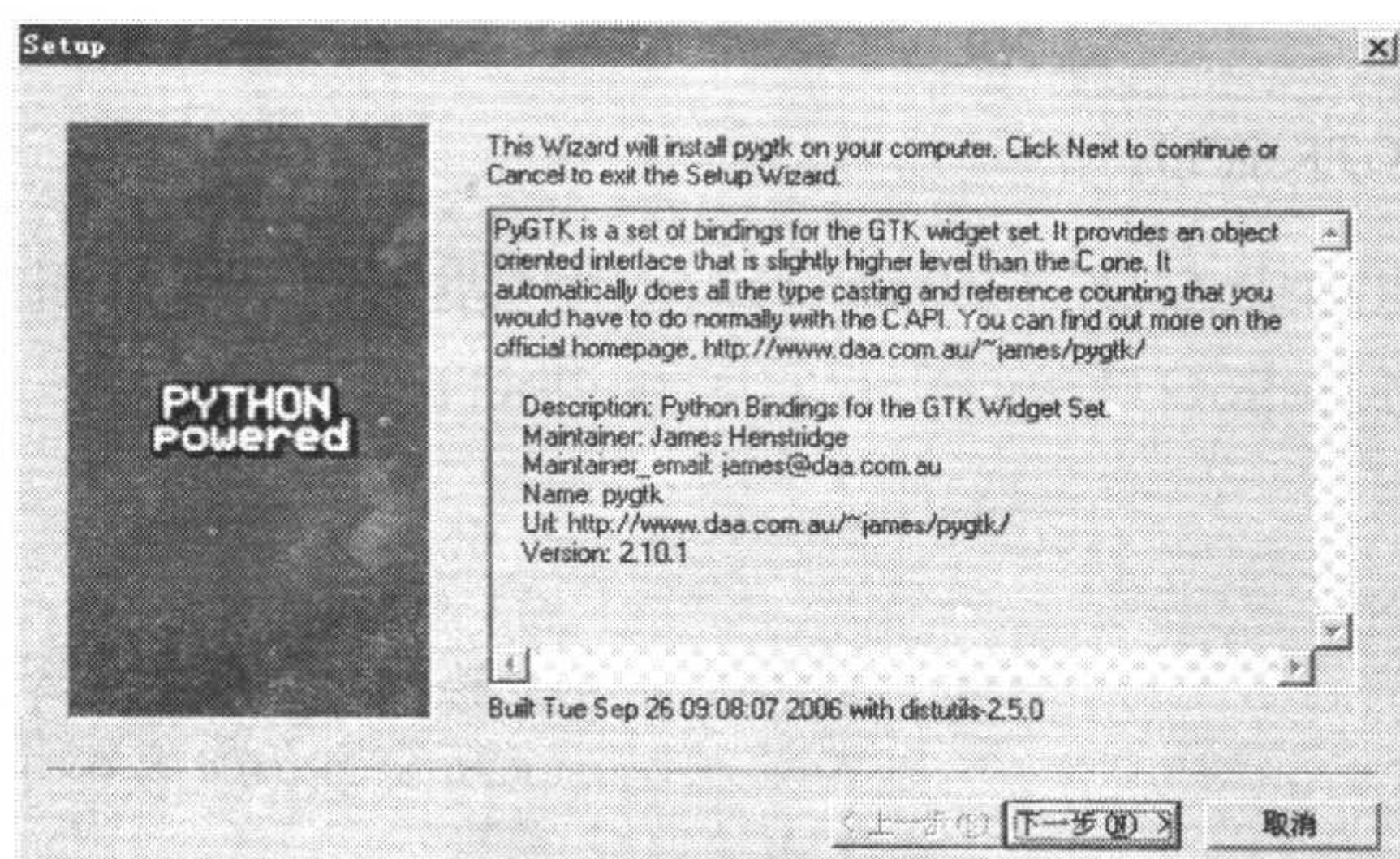


图 14-3 安装 pygtk 模块

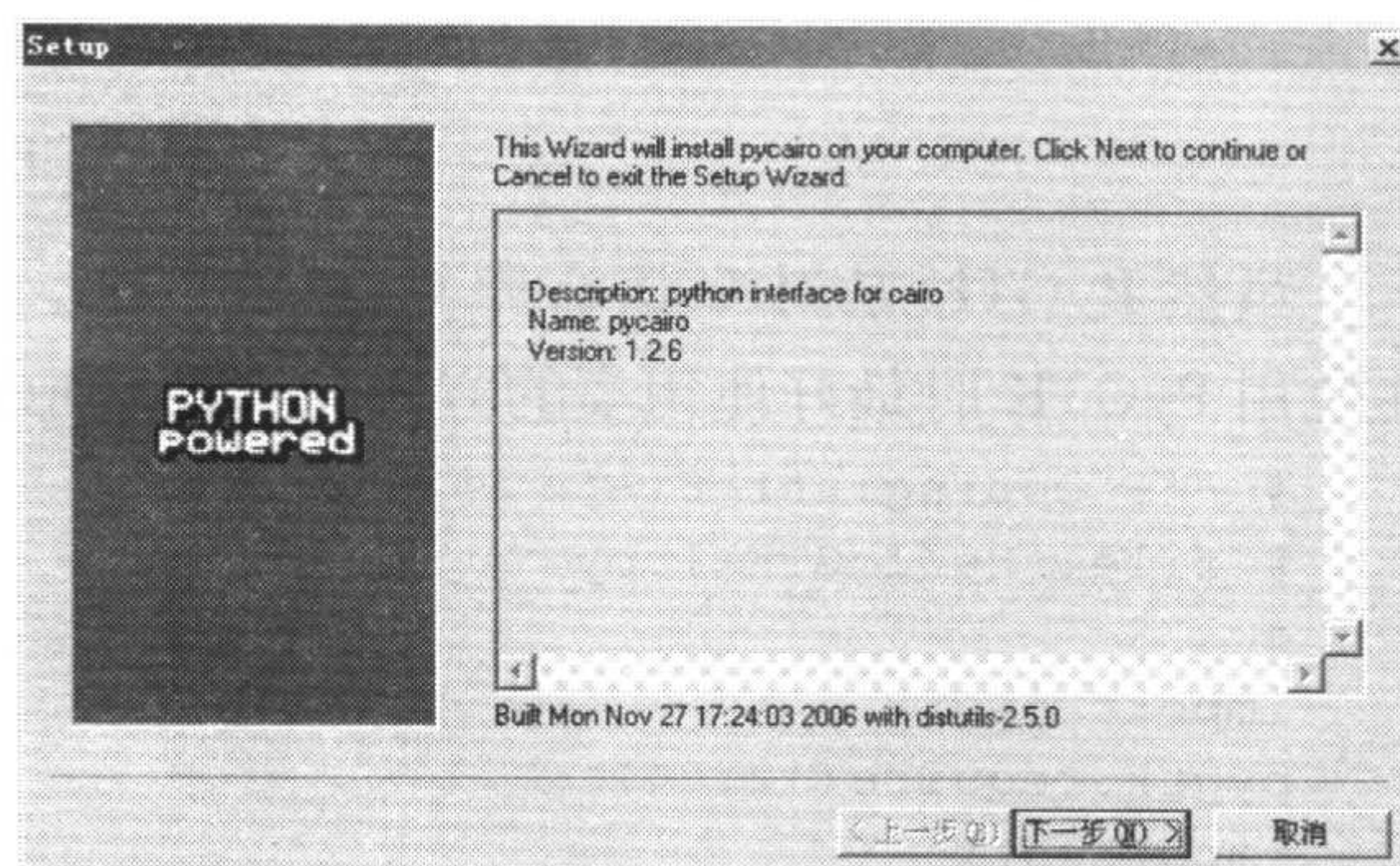


图 14-4 安装 pycairo 模块

14.1.2 创建窗口

使用 gtk 模块中的 `gtk.Window` 类可以创建一个窗口。窗口的属性，如大小、标题等，需要调用窗口对象的相应方法进行设置。当完成窗口设置后，可以调用窗口对象的 `show` 方法显示窗口，然后调用 `gtk.main` 函数进入消息循环。常用的窗口属性设置方法有以下几种。

- `set_title(title)`: 设置窗口标题。
- `set_position(position)`: 设置窗口位置。
- `fullscreen()`: 设置为全屏窗口。
- `set_default_size(width, height)`: 设置窗口的默认大小。

如下所示的 `HelloGTK.py` 脚本使用 `gtk.Window()` 创建了一个 GUI 窗口。

```
# -*- coding:utf-8 -*-
# file: HelloGTK.py
#
import pygtk
```

```
# 导入 pygtk 模块
```



```

pygtk.require('2.0')
import gtk
window = gtk.Window()
window.set_title('PyGTK')
window.set_default_size(300, 200)
window.show()
gtk.main()

```

设置 pygtk 所需的 gtk 版本
导入 gtk 模块
创建窗口对象
设置窗口标题
设置窗口大小
显示窗口
进入消息循环

运行 HelloGTK.py 脚本将创建如图 14-5 所示的窗口。

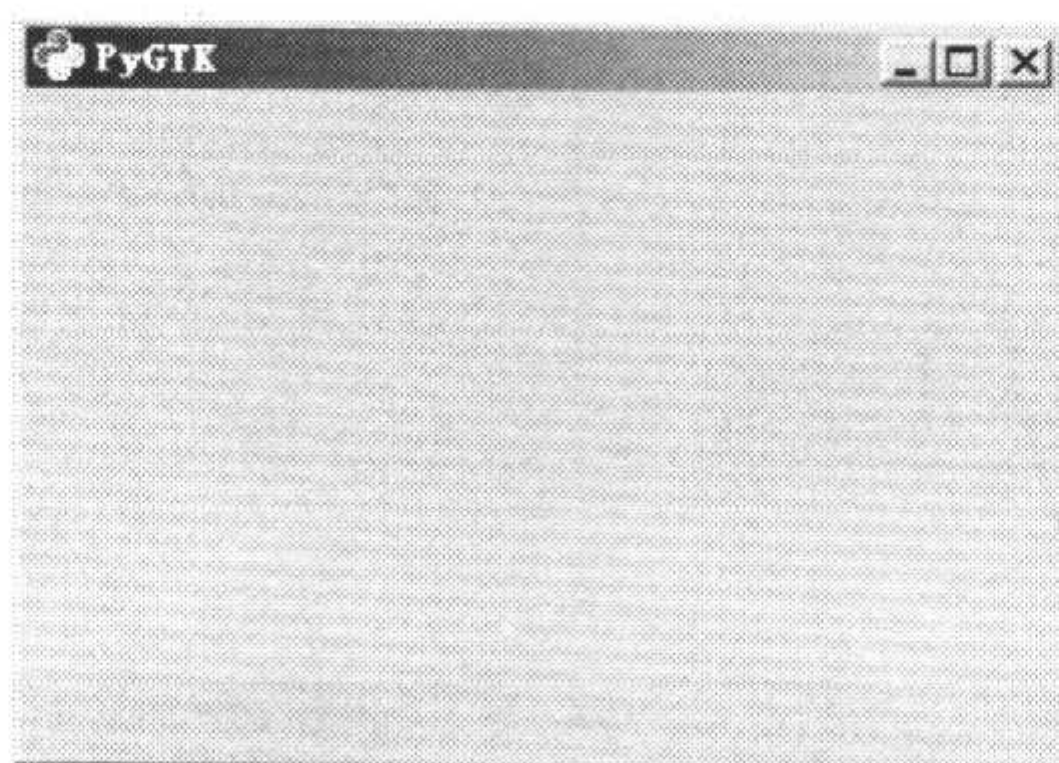


图 14-5 PyGTK 窗口

虽然 PyGTK 是对 C 语言库 GTK 的封装，但由于 GTK 在设计时就使用了类的思想，因此在使用 PyGTK 时最好使用类的方式。如下所示的 HelloGTK++.py 改写了 HelloGTK.py 脚本。

```

# -*- coding:utf-8 -*-
# file: HelloGTK++.py
#
import pygtk
pygtk.require('2.0')
import gtk
class MyWindow():
    def _init_(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.show()
    def main(self):
        gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
显示窗口
定义 main 方法
调用 gtk.main 方法
创建窗口对象
进入消息循环

14.2 组件

PyGTK 提供了许多常用组件，这些组件的风格可能与 Windows 下常见的组件风格不同。在 PyGTK 中创建组件略微繁琐，组件创建后往往要调用组件对象的方法才能设置组件的属性。

14.2.1 标签

PyGTK 中的标签用于显示文本。当创建标签以后，可以调用标签对象的方法设置标签中

的文字、文字的对齐方式、标签的角度等。

1. 创建标签

使用 `gtk.Label` 可以创建标签并设置标签的文本。标签对象具有以下几种方法用于设置，或者获取标签的属性。

- `set_text()`: 重新设置标签文本。
- `get_text()`: 获取标签文本。
- `set_justify()`: 设置标签中文本的对齐方式。
- `get_justify()`: 获取标签中文本的对齐方式。
- `set_angle()`: 设置标签的角度，角度值应为 90 或 270。
- `get_angle()`: 获取标签的角度。

如下所示的 `PyGTKLabel.py` 使用 `gtk.Label` 创建标签。

```
# -*- coding:utf-8 -*-
# file: PyGTKLabel.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        label = gtk.Label('PyGTK')
        label.set_angle(90)
        self.window.add(label)
        label.show()
        self.window.show()
    def main(self):
        gtk.main()

window = MyWindow('PyGTK', 300, 200)
window.main()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
创建标签
设置标签角度
向窗口中添加标签
显示标签
显示窗口
定义 main 方法
调用 gtk.main 方法
创建窗口对象

运行 `PyGTKLabel.py` 脚本后将创建如图 14-6 所示的窗口。



图 14-6 创建标签

2. 设置 PyGTK 默认字体

运行 PyGTKLabel.py 脚本后，如果在命令行窗口中出现无法载入字体的警告，可以通过设置 PyGTK 的默认字体来解决。假设 PyGTK 的安装目录为“C:\Program Files\PyGTK”，使用文本编辑器打开“C:\Program Files\PyGTK\GTK\etc\gtk-2.0”目录下的“gtkrc”文件。其内容如下所示。

```
gtk-theme-name = "MS-Windows"
```

将其修改为如下所示的内容。

```
style "user-font"
{
    font_name="Tahoma, SimSun 10"
}
widget_class "*" style "user-font"
gtk-font-name = "Tahoma, Simsun 10"
gtk-theme-name = "MS-Windows"
```

如果需要在 PyGTK 的组件中使用中文，除了在脚本的首行使用“# -*- coding:utf-8 -*-”以外，还应该将脚本保存成 UTF-8 的编码格式。

3. 向窗口中添加多个标签

由于 gtk.window 对象每次只能容纳一个组件。如果需要向其中添加多个组件，则需要使用 Box 对象。PyGTK 提供了 gtk.HBox()和 gtk.VBox()用于创建 Box 对象，它们可以相互嵌套。当创建 Box 对象后，可以使用其 pack_start 和 pack_end 方法向其中添加组件。其原型分别如下所示。

```
pack_start(child, expand=True, fill=True, padding=0)
pack_end(child, expand=True, fill=True, padding=0)
```

其参数含义如下。

- child: 向 Box 对象中添加的组件。
- expand: Bool 型，是否允许组件获取更大的空间。
- fill: Bool 型，只有在 expand 为 True 时才有效。
- padding: Box 对象中组件间的距离。

对于 pack_start 方法，其每次添加的组件位于所有使用 pack_start 方法添加的组件之后。对于 pack_end 方法，其每次添加的组件位于所有使用 pack_end 方法添加的组件之前。如下所示的 PyGTKLabelM.py 脚本使用 Box 对象向窗口中添加多个标签。

```
# -*- coding:utf-8 -*-
# file: PyGTKLabelM.py
#
import pygtk
pygtk.require('2.0')
import gtk
class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
```

```
# 导入 pygtk 模块
# 设置 pygtk 所需的 gtk 版本
# 导入 gtk 模块
# 定义窗口类
# 定义初始化方法
# 生成窗口对象
```



```

self.window.set_title(title)
self.window.set_default_size(width, height)
vbox = gtk.VBox(False, 5)
hbox1 = gtk.HBox(False, 5)
hbox2 = gtk.HBox(False, 5)
label1 = gtk.Label('Label1')
label1.set_angle(90)
label2 = gtk.Label('Label2')
label2.set_angle(270)
label3 = gtk.Label('Label3')
label4 = gtk.Label('Label4')
label5 = gtk.Label('Label5')
hbox1.pack_start(label1)
hbox1.pack_start(label2)
hbox2.pack_start(label3)
hbox2.pack_end(label4)
hbox2.pack_end(label5)
vbox.pack_start(hbox1)
vbox.pack_start(hbox2)
self.window.add(vbox)
label1.show()
label2.show()
label3.show()
label4.show()
label5.show()
hbox1.show()
hbox2.show()
vbox.show()
self.window.show()

def main(self):
    gtk.main()

window = MyWindow('PyGTK', 300, 200)
window.main()

```

设置窗口标题
设置窗口大小
生成竖向 Box 对象
生成水平 Box 对象

创建标签
设置标签角度

向 Box 对象中添加标签

向 Box 对象中添加其他 Box 对象

向窗口中添加 Box 对象
显示标签

显示 Box 对象

显示窗口
定义 main 方法
调用 gtk.main 方法
创建窗口对象

运行 PyGTKLabelM.py 脚本后将创建如图 14-7 所示的窗口。

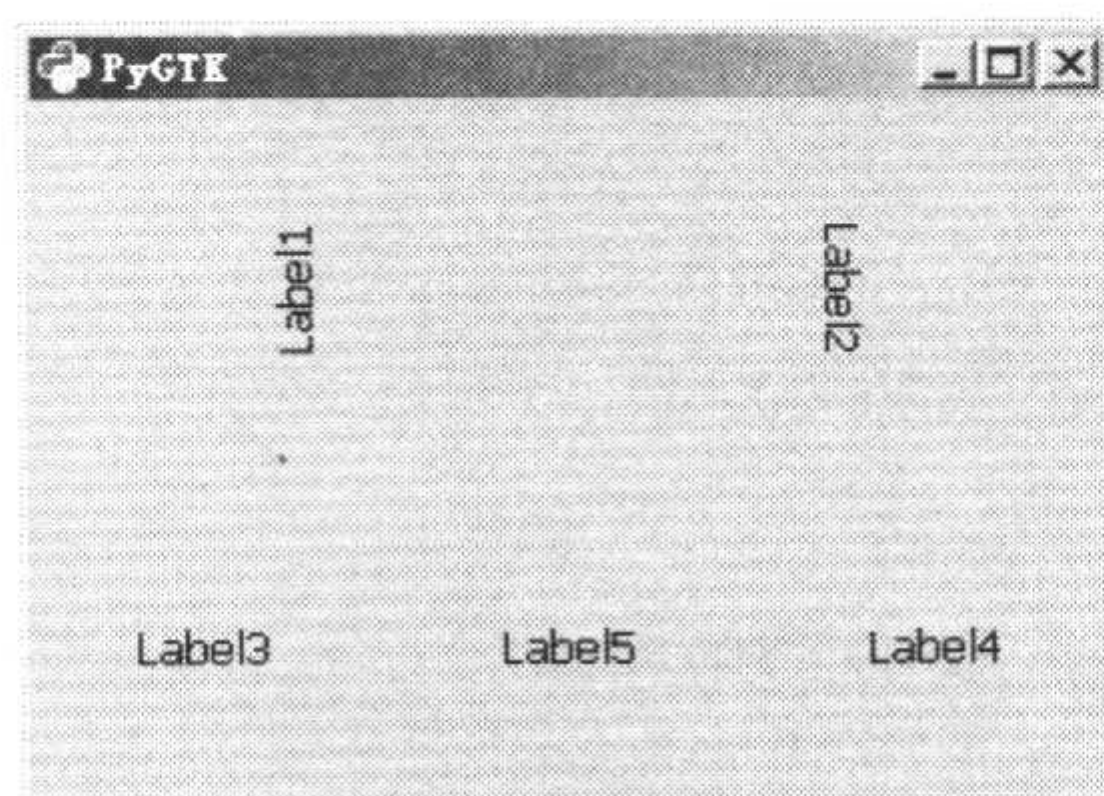


图 14-7 创建多个标签

14.2.2 按钮

在 PyGTK 中使用 `gtk.Button` 可以创建按钮。通过按钮的 `connect` 方法可以将按钮事件信

号绑定到事件处理函数上。

1. 创建按钮

使用 `gtk.Button` 创建按钮对象后，可以使用以下几个按钮对象的方法设置按钮属性，或者获取按钮的属性。

- `pressed()`：产生 `pressed` 信号。
- `released()`：产生 `released` 信号。
- `clicked()`：产生 `clicked` 信号。
- `enter()`：产生 `enter` 信号。
- `leave()`：产生 `leave` 信号。
- `set_label()`：设置按钮文字。
- `get_label()`：获得按钮文字。

如下所示的 `PyGTKButton.py` 创建了两个按钮。

```
# -*- coding:utf-8 -*-
# file: PyGTKButton.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        hbox = gtk.HBox(False, 20)
        button1 = gtk.Button('Button1')
        button2 = gtk.Button('Button2')
        hbox.pack_start(button1)
        hbox.pack_start(button2)
        self.window.add(hbox)
        hbox.show()
        button1.show()
        button2.show()
        self.window.show()
    def main(self):
        gtk.main()

window = MyWindow('PyGTK', 150, 30)
window.main()
```

导入 pygtk 模块
 # 设置 pygtk 所需的 gtk 版本
 # 导入 gtk 模块
 # 定义窗口类
 # 定义初始化方法
 # 生成窗口对象
 # 设置窗口标题
 # 设置窗口大小
 # 生成水平 Box 对象
 # 创建按钮
 # 向 Box 对象中添加按钮
 # 向窗口添加 Box 对象
 # 显示 Box 对象
 # 显示按钮
 # 显示窗口
 # 定义 main 方法
 # 调用 gtk.main 方法
 # 创建窗口对象

运行 `PyGTKButton.py` 脚本后将创建如图 14-8 所示的窗口。

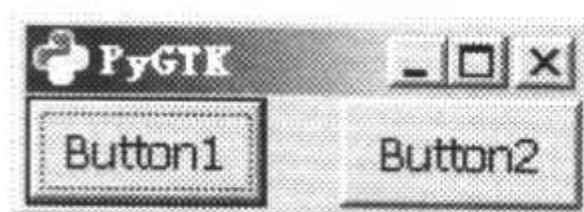


图 14-8 创建按钮

2. 按钮事件

绑定按钮事件可以使用按钮对象的 `connect` 方法，其原型如下所示。

```
connect(name, func, func_data)
```

其参数含义如下。

- `name`: 事件信号名，对于按钮可以是 `clicked`、`pressed` 等。
- `func`: 事件响应函数。
- `func_data`: 可选参数，传递给事件相应函数的附加数据。

其中，事件响应函数应该根据 `connect` 函数所使用的参数来定义。如果在 `connect` 函数中使用了 `func_data` 参数，则事件响应函数应声明成如下形式。

```
def func(self, widget, data):
    <处理语句>
```

如果在 `connect` 函数中没有使用 `func_data` 参数，则事件响应函数应声明成如下所示的形式。

```
def func(self, widget):
    <处理语句>
```

如下所示的 `PyGTKButtonEvent.py` 使用按钮对象的 `connect` 方法绑定按钮事件。

```
# -*- coding:utf-8 -*-
# file: PyGTKButtonEvent.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())

        hbox = gtk.HBox(False, 20)
        self.button1 = gtk.Button('Button1')
        self.button2 = gtk.Button('Button2')
        self.button1.connect('clicked', self.OnButton1, 'Button1')

        self.button2.connect('clicked', self.OnButton2, 'Button2')
        hbox.pack_start(self.button1)
        hbox.pack_start(self.button2)
        self.window.add(hbox)
        hbox.show()
        self.button1.show()
        self.button2.show()
        self.window.show()

    def main(self):
        gtk.main()

    def OnButton1(self, widget, data):
        # 导入 pygtk 模块
        # 设置 pygtk 所需的 gtk 版本
        # 导入 gtk 模块
        # 定义窗口类
        # 定义初始化方法
        # 生成窗口对象
        # 设置窗口标题
        # 设置窗口大小
        # 关闭窗口时退出程序
        # 生成水平 Box 对象
        # 创建按钮
        # 绑定按钮事件
        # 向 Box 对象中添加按钮
        # 向窗口中添加 Box 对象
        # 显示 Box 对象
        # 显示按钮
        # 显示窗口
        # 定义 main 方法
        # 调用 gtk.main 方法
        # 处理按钮事件
```

```

        self.button2.set_label('Quit')
    def OnButton2(self, widget, data):
        gtk.main_quit()
window = MyWindow('PyGTK', 150, 30)
window.main()

```

重新设置 Button2 文本
处理按钮事件
退出程序
创建窗口对象

运行 PyGTKButtonEvent.py 脚本后，单击【Button1】按钮可以改变【Button2】的标签。单击【Button2】则可以退出窗口。另外，脚本中还使用 window 对象的 connect 绑定了“destroy”信号，之前的脚本关闭窗口后，脚本并不退出。这是因为关闭窗口后仅发送“destroy”信号而没有退出 gtk.main()，而在该脚本中使用 lambda 将 window 对象的“destroy”信号绑定到 gtk.main_quit() 函数，这样当窗口关闭后就可以退出脚本了。

14.2.3 容器组件

使用 Box 对象放置组件不能调整组件的大小，也不能任意地放置组件。对于比较复杂的 GUI 使用 Box 对象显然不能满足要求。在 PyGTK 中还有容器组件用于放置其他组件。例如，Fixed 组件和 Layout 组件，它们都可以使用坐标的形式布置组件。

1. Fixed 组件

使用 gtk.Fixed() 可以创建一个 Fixed 组件，通过 Fixed 组件对象的 put 方法可以向 Fixed 组件中添加其他组件。而使用 move 方法则可以改变组件在 Fixed 组件中的位置。其原型分别如下所示。

```

put(widget, x, y)
move(widget, x, y)

```

其参数含义相同，如下所示。

- widget: 要添加或者移动的组件对象。
- x: x 坐标值。
- y: y 坐标值。

如下所示的 PyGTKFixed.py 脚本使用 Fixed 组件向窗口中添加多个组件。

```

# -*- coding:utf-8 -*-
# file: PyGTKFixed.py
#
import pygtk
pygtk.require('2.0')
import gtk
class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('PyGTK')

```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
创建标签


```

self.fixed.put(self.label, 10, 5)
self.button = gtk.Button('Move')
self.button.connect('clicked', self.OnButton, 'Move')
self.fixed.put(self.button, 120, 150)
self.window.add(self.fixed)
self.label.show()
self.button.show()
self.fixed.show()
self.window.show()
def OnButton(self, widget, data):
    self.fixed.move(self.label, 100, 50)
def main(self):
    gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

添加标签
创建按钮
绑定按钮事件
添加按钮
向窗口中添加 Fixed
显示标签
显示按钮
显示 Fixed 组件
显示窗口
定义 main 方法
创建窗口对象

运行 PyGTKFixed.py 脚本后将创建如图 14-9 所示的窗口。单击【Move】按钮，标签将移动到另一位置，如图 14-10 所示。

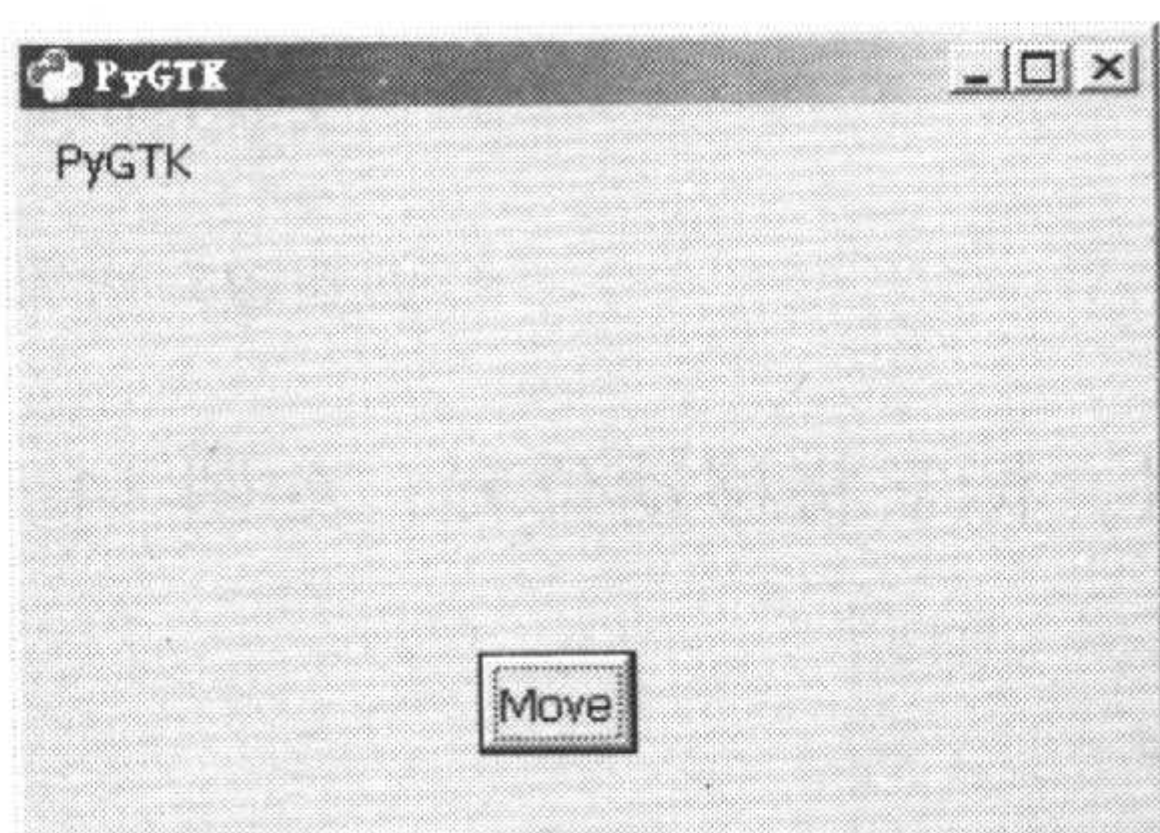


图 14-9 使用 Fixed 组件

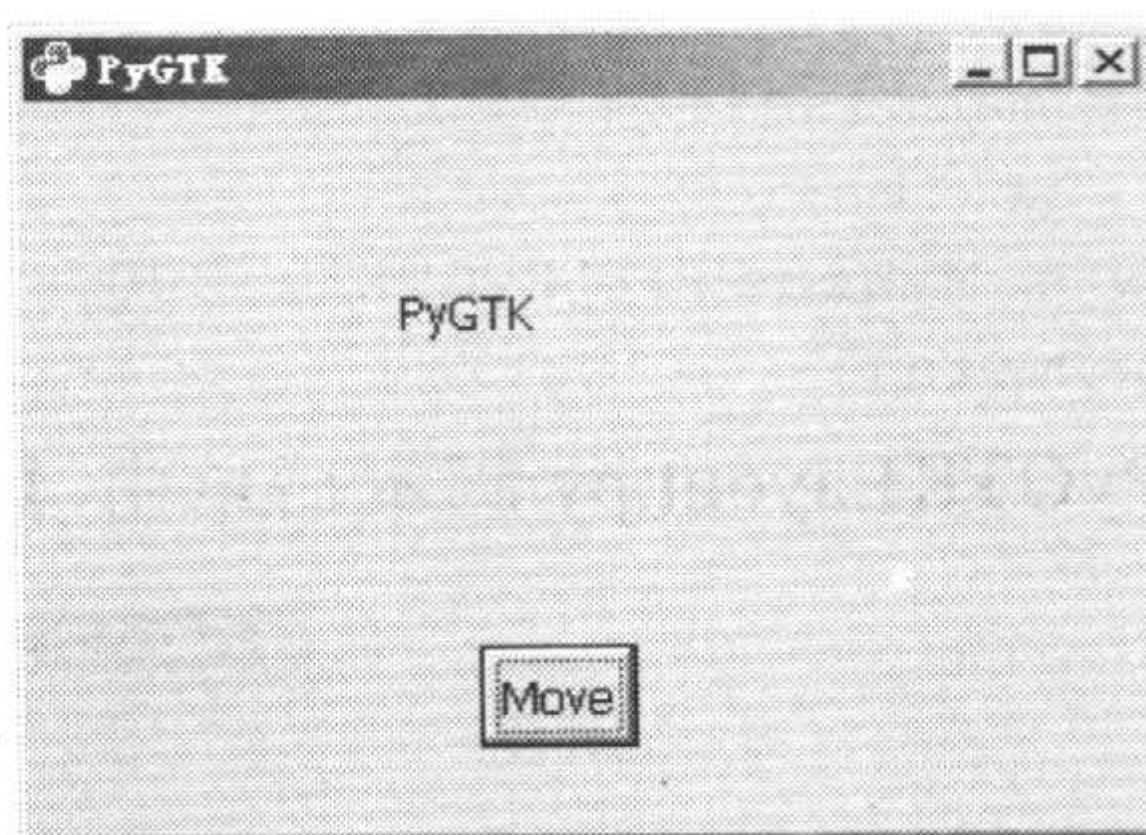


图 14-10 移动后的标签位置

2. Layout 组件

Layout 组件和 Fixed 组件基本相同，它也具有 put 方法和 move 方法可以添加或者移动组件的位置。但 Layout 可以创建更大的范围。如下所示的 PyGTKLayout.py 脚本使用 Layout 组件向窗口中添加多个组件。

```

# -*- coding:utf-8 -*-
# file: PyGTKLayout.py
#
import pygtk
pygtk.require('2.0')
import gtk
class MyWindow():
    def __init__(self, title, width, height):
        self.x = 10
        self.y = 5
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())

```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
定义坐标信息
生成窗口对象
设置窗口标题
设置窗口大小


```

self.layout = gtk.Layout()
self.label = gtk.Label('PyGTK')
self.layout.put(self.label, self.x, self.y)
self.button = gtk.Button('Move')
self.button.connect('clicked', self.OnButton, 'Move')
self.layout.put(self.button, 120, 150)
self.window.add(self.layout)
self.label.show()
self.button.show()
self.layout.show()
self.window.show()
def OnButton(self, widget, data):
    self.x = self.x + 5
    self.y = self.y + 5
    if self.x >= 300:
        self.x = 10
    if self.y >= 200:
        self.y = 5
    self.layout.move(self.label, self.x, self.y)
def main(self):
    gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

创建标签
添加标签
创建按钮
绑定按钮事件
添加按钮
向窗口中添加 Layout
显示标签
显示按钮
显示 Layout 组件
显示窗口
按钮事件响应函数
移动标签
定义 main 方法
创建窗口对象

运行 PyGTKLayout.py 脚本后单击 **【Move】** 按钮，标签将移动位置，如图 14-11 所示。

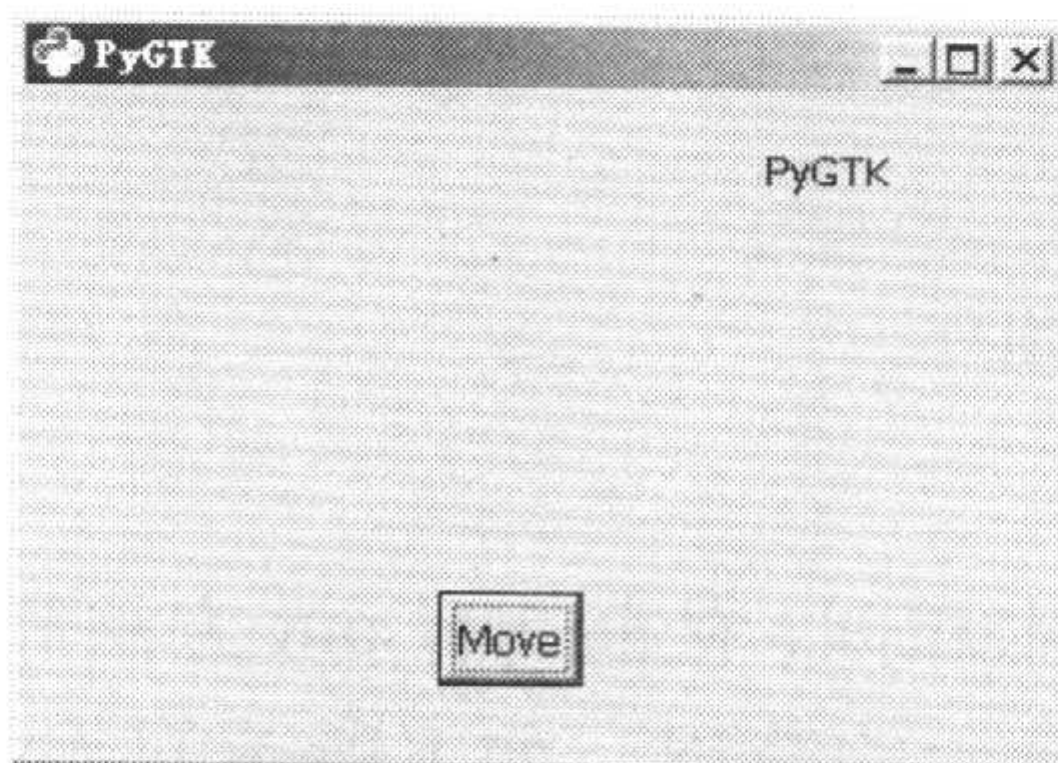


图 14-11 使用 Layout 组件

14.2.4 文本框

PyGTK 中单行文本框的创建相对较简单，但创建多行文本框则有点繁琐。多行文本框中的内容需要使用 `gtk.TextBuffer` 对象进行设置或者操作，而且多行文本框一般需要同滚动窗口组合使用。

1. 单行文本框

使用 `gtk.Entry()` 可以创建单行文本框。单行文本框具有以下几种常用的方法。

- `set_visibility()`: 将单行文本框设置为密码框。
- `get_visibility()`: 判断单行文本框是否为密码框。
- `set_max_length()`: 设置单行文本框所能输入的最长字符数。

- `get_max_length()`: 获取单行文本框所能输入的最长字符数。
- `set_text()`: 设置单行文本框中的文字。
- `get_text()`: 获取单行文本框中的文字。

如下所示的 `PyGTKEntry.py` 脚本使用 `gtk.Entry()` 创建了一个单行文本框和密码框。

```
# -*- coding:utf-8 -*-
# file: PyGTKEntry.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def _init_(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())

        vbox = gtk.VBox(False, 5)
        label1 = gtk.Label('Nomal')
        vbox.pack_start(label1)
        entry1 = gtk.Entry()
        vbox.pack_start(entry1)
        entry1.show()
        label2 = gtk.Label('Password')
        vbox.pack_start(label2)
        entry2 = gtk.Entry()
        entry2.set_visibility(False)
        vbox.pack_start(entry2)
        entry2.show()
        self.window.add(vbox)
        label1.show()
        label2.show()
        vbox.show()
        self.window.show()

    def main(self):
        gtk.main()

window = MyWindow('PyGTK', 200, 120)
window.main()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
关闭窗口时退出程序
生成竖向 Box 对象
创建标签
向 Box 对象中添加标签
创建文本框
向 Box 对象中添加文本框
显示文本框
创建标签
创建文本框
将文本框设置为密码框
向窗口中添加 Box 对象
显示标签
显示窗口
定义 main 方法
调用 gtk.main 方法
创建窗口对象

运行 `PyGTKEntry.py` 脚本后在单行文本框和密码框中输入文字，如图 14-12 所示。

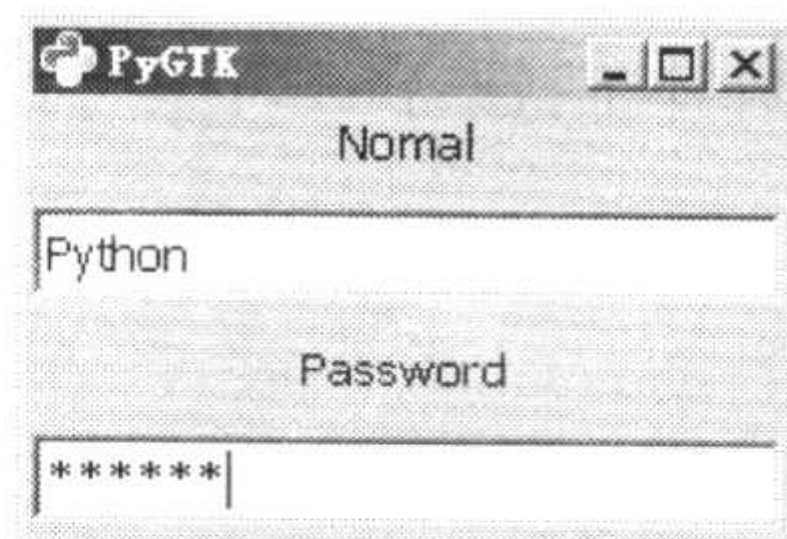


图 14-12 单行文本框和密码框

2. 多行文本框

使用 PyGTK 中的 `gtk.TextView` 创建多行文本框比较复杂。由于 PyGTK 中的多行文本框不含滚动条，因此对于多行文本框常将其添加到一个滚动窗口中，使用滚动窗口的滚动条。当文本框创建后，应使用其 `get_buffer` 方法创建 `gtk.TextBuffer` 对象。使用 `gtk.TextBuffer` 对象可以获得文本框中的内容，或者对文本框中的内容进行操作。`gtk.TextView` 常用的方法有以下几个。

- `set_buffer()`: 设置 `gtk.TextBuffer` 对象。
- `get_buffer()`: 获得 `gtk.TextBuffer` 对象。
- `set_wrap_mode()`: 设置换行方式。
- `get_wrap_mode()`: 获得换行方式。
- `set_editable()`: 设置文本框中文本的可编辑性。
- `get_editable()`: 获得文本框中文本是否可编辑。
- `set_justification()`: 设置对齐方式。
- `get_justification()`: 获得对齐方式。
- `set_left_margin()`: 设置左边距。
- `get_left_margin()`: 获得左边距。
- `set_right_margin()`: 设置右边距。
- `get_right_margin()`: 获得右边距。
- `set_tabs()`: 设置制表符大小。
- `get_tabs()`: 获取制表符大小。

`gtk.TextBuffer` 常用的方法有以下几个。

- `get_line_count()`: 获取 `TextBuffer` 中的行数。
- `get_char_count()`: 获取 `TextBuffer` 中的字符数。
- `set_text()`: 设置 `TextBuffer` 中的文本。
- `insert()`: 插入文本。
- `insert_at_cursor()`: 在光标处插入文本。
- `get_text()`: 获得 `TextBuffer` 中的文本。

如下所示的 `PyGTKTextView.py` 脚本创建了一个多行文本框。

```
# -*- coding:utf-8 -*-
# file: PyGTKTextView.py
#
import pygtk
pygtk.require('2.0')
import gtk

# 导入 pygtk 模块
# 设置 pygtk 所需的 gtk 版本
# 导入 gtk 模块
```

第14章 使用PyGTK编写GUI

```

class MyWindow():
    def _init_(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())

        vbox = gtk.VBox(False, 5)
        swindow = gtk.ScrolledWindow()
        text = gtk.TextView()
        textbuffer = text.get_buffer()
        swindow.add(text)
        swindow.show()
        vbox.pack_start(swindow)
        text.show()
        self.window.add(vbox)
        vbox.show()
        self.window.show()

    def main(self):
        gtk.main()

window = MyWindow('PyGTK', 300, 200)
window.main()

```

定义窗口类
 # 定义初始化方法
 # 生成窗口对象
 # 设置窗口标题
 # 设置窗口大小
 # 关闭窗口时退出程序
 # 生成竖向 Box 对象
 # 创建多行文本框
 # 文本框缓冲区
 # 向 Box 对象中添加文本框
 # 显示文本框
 # 向窗口添加 Box 对象
 # 显示窗口
 # 定义 main 方法
 # 调用 gtk.main 方法
 # 创建窗口对象

运行 PyGTKTextView.py 脚本后将创建如图 14-13 所示的多行文本框。在多行文本框中已经实现了右键菜单，在窗口中单击鼠标右键将出现如图 14-14 所示的快捷菜单。

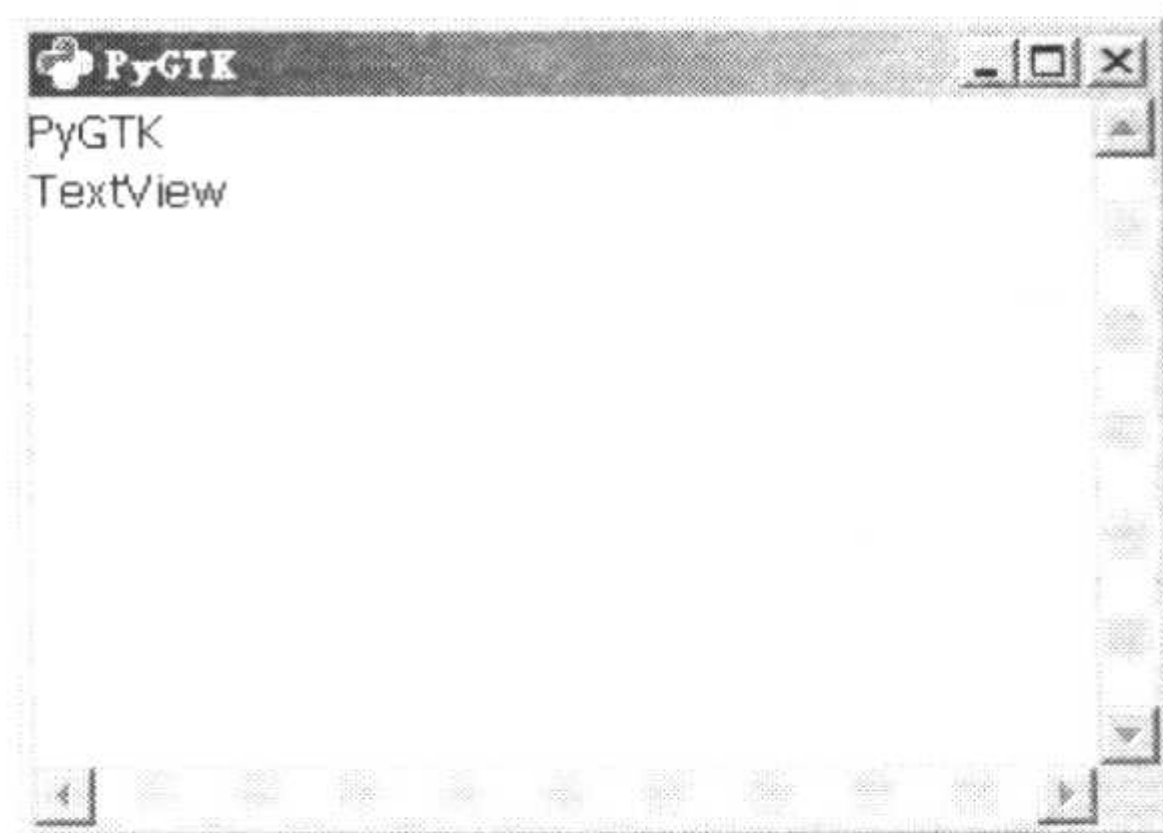


图 14-13 多行文本框

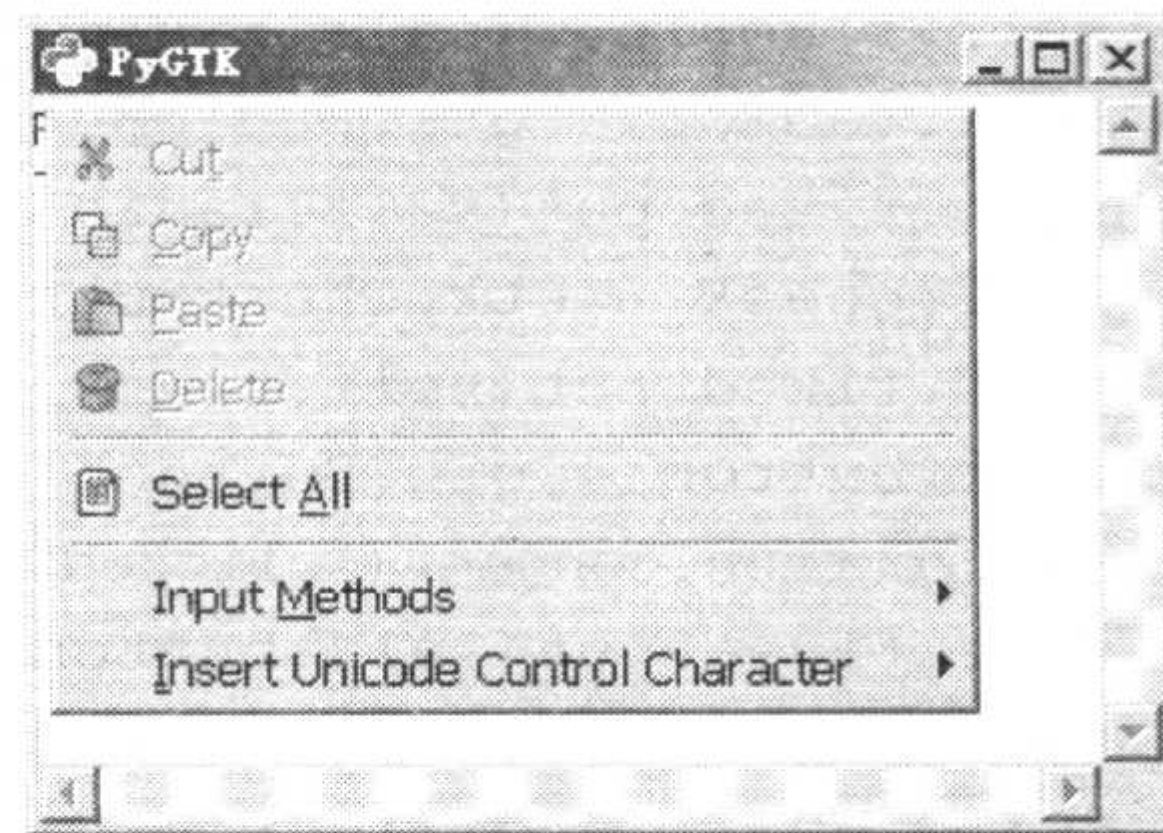


图 14-14 文本框中的右键菜单

14.2.5 单选框和复选框

使用 `gtk.RadioButton` 可以创建单选框，其具有以下几个初始化参数。

- `group`: 单选框所在的组。
- `label`: 单选框显示的文本。
- `use_underline`: 可选参数，若为真，表示设置单选框文本中位于下划线之后的字母为快捷键。

当使用 `gtk.RadioButton` 创建单选框对象后，可以使用单选框对象的 `get_active` 方法判断

单选框是否被选中。使用 `gtk.CheckButton` 可以创建复选框，它不具有 `group` 初始化参数，其余初始化参数与 `gtk.RadioButton` 的初始化参数相同。当使用 `gtk.CheckButton` 创建复选框对象后，也可以使用复选框对象的 `get_active` 方法判断复选框是否被选中。如下所示的 `PyGTKRCbutton.py` 脚本创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: PyGTKRCbutton.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def _init_(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label1 = gtk.Label('PyGTK')
        self.fixed.put(self.label1, 80, 20)
        self.label2 = gtk.Label('PyGTK')
        self.fixed.put(self.label2, 160, 20)
        self.radio1 = gtk.RadioButton(None, 'Radio1')
        self.fixed.put(self.radio1, 50, 60)
        self.radio2 = gtk.RadioButton(self.radio1, 'Radio2')
        self.fixed.put(self.radio2, 50, 90)
        self.radio3 = gtk.RadioButton(self.radio1, 'Radio3')
        self.fixed.put(self.radio3, 50, 120)
        self.check = gtk.CheckButton('CheckButton')
        self.fixed.put(self.check, 150, 60)
        self.button = gtk.Button('Test')
        self.button.connect('clicked', self.OnButton, 'Test')
        self.fixed.put(self.button, 120, 150)
        self.window.add(self.fixed)
        self.label1.show()
        self.label2.show()
        self.radio1.show()
        self.radio2.show()
        self.radio3.show()
        self.check.show()
        self.button.show()
        self.fixed.show()
        self.window.show()

    def OnButton(self, widget, data):
        if self.check.get_active():
            self.label2.set_text('checked')
        else:
            self.label2.set_text('uncheck')
        if self.radio1.get_active():
            # 判断复选框是否被选中
            # 判断复选框选中状态
```



```

        self.label1.set_text('Radio1')
    elif self.radio2.get_active():
        self.label1.set_text('Radio2')
    else:
        self.label1.set_text('Radio3')
def main(self):
    gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

定义 main 方法

创建窗口对象

运行 PyGTKRCbutton.py 脚本后, 单击【Test】按钮, 将根据单选框和复选框的状态分别设置按钮标签文字, 如图 14-15 所示。

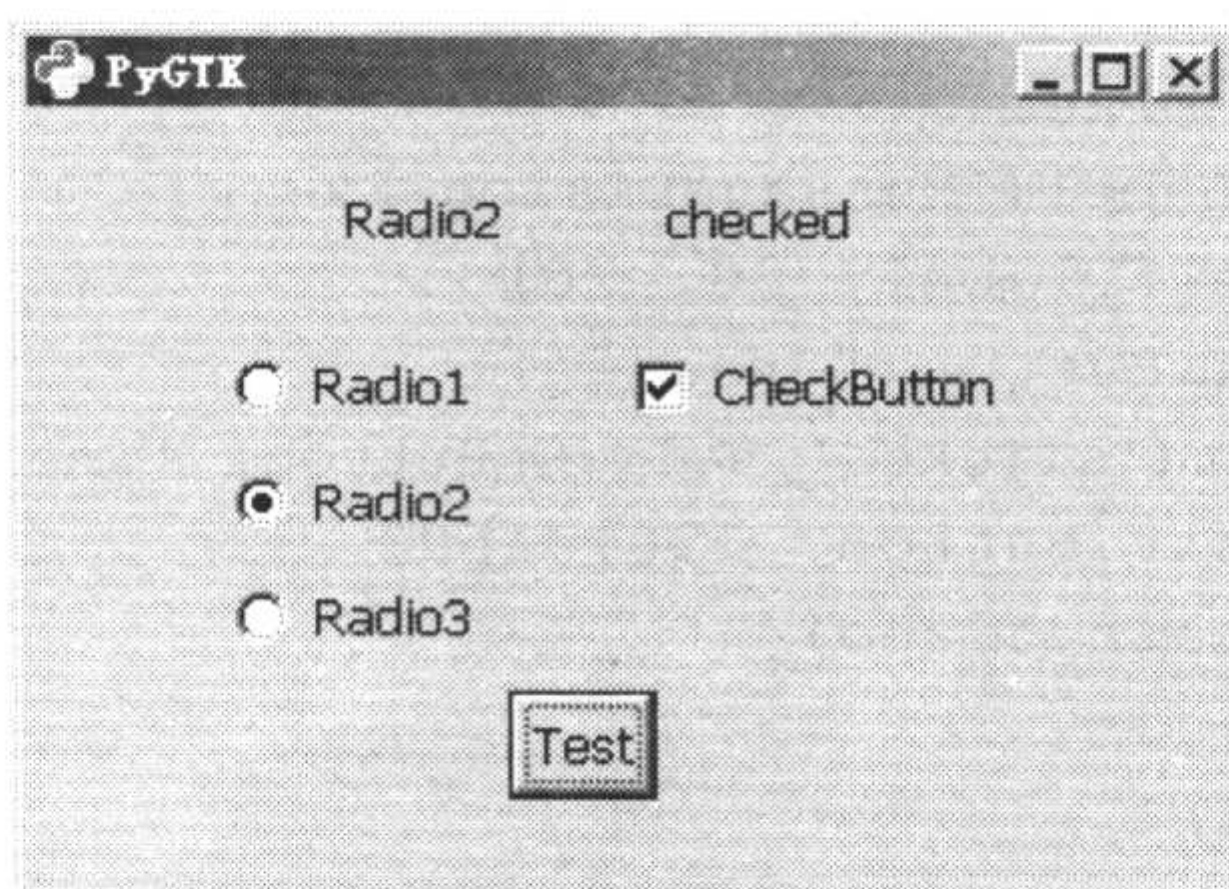


图 14-15 单选框和复选框

14.3 消息框和对话框

与前边所使用的 GUI 编程工具包不同, PyGTK 提供的消息框和对话框风格是 Linux 下的风格, 与 Windows 系统的风格不一样。

14.3.1 消息框

使用 `gtk.MessageDialog` 可以创建消息框, 当创建消息框以后需要调用消息框对象的 `run` 方法才能显示消息框。最后还应该调用消息框的 `destroy` 方法销毁消息框。`gtk.MessageDialog` 的原型如下所示。

```
gtk.MessageDialog(parent, flags, type, buttons_NONE, message)
```

其参数含义如下所示。

- parent: 消息框的父窗口, 可以为 None。
- flags: 消息框的创建标志, 可以为 0。
- type: 消息框类型。
- buttons: 消息框中的按钮。
- message: 消息框中所显示的文本。

如下所示的 PyGTKMessage.py 使用 gtk.MessageDialog 创建消息框。

```
# -*- coding:utf-8 -*-
# file: PyGTKMessage.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def _init_(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('MessageDialog Example')
        self.fixed.put(self.label, 80, 20)
        self.button = gtk.Button('Create')
        self.button.connect('clicked', self.OnButton, 'Create')
        self.fixed.put(self.button, 120, 150)
        self.window.add(self.fixed)
        self.label.show()
        self.button.show()
        self.fixed.show()
        self.window.show()

    def OnButton(self, widget, data):
        msg = gtk.MessageDialog(self.window,
                                gtk.DIALOG_MODAL,
                                gtk.MESSAGE_INFO,
                                gtk.BUTTONS_OK,
                                'An example')
        msg.run()
        msg.destroy()

    def main(self):
        gtk.main()

window = MyWindow('PyGTK', 300, 200)
window.main()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
创建标签
添加标签
创建按钮
绑定按钮事件
添加按钮
向窗口中添加 Fixed
显示组件
按钮事件处理函数
创建消息框
消息框标志
消息框类型
消息框按钮
消息框中的内容
显示消息框
销毁消息框
定义 main 方法
创建窗口对象

运行 PyGTKMessage.py 脚本后单击 **【Create】** 按钮，将创建如图 14-16 所示的消息框。

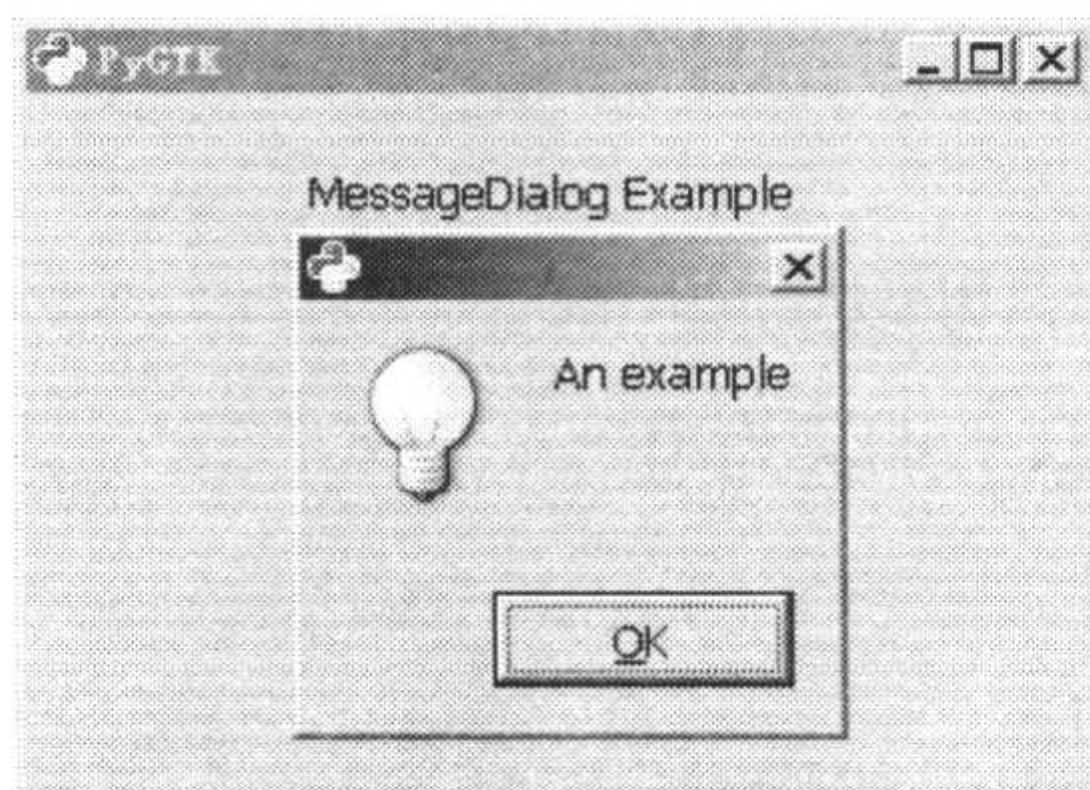


图 14-16 创建消息框

14.3.2 标准对话框

PyGTK 中提供了标准的文件打开、保存对话框、颜色选择对话框、字体选择对话框等。使用 `gtk.FileChooserDialog` 可以创建文件打开、保存对话框。使用 `gtk.ColorSelectionDialog` 可以创建颜色选中对话框。使用 `gtk.FontSelectionDialog` 可以创建字体选择对话框。

使用 `gtk.FileChooserDialog` 时，可以使用 `gtk.FileFilter` 过滤文件，即只查看某一类型的文件。`gtk.FileFilter` 有以下常用方法。

- `set_name()`: 设置文件类型名。
- `get_name()`: 获得文件类型名。
- `add_pattern()`: 文件类型的后缀，如 Python 文件则写成 “*.py” 的形式。

当创建好 `gtk.FileFilter` 后，可以使用文件打开、关闭对话框对象的 `add_filter` 对象将其添加到对话框中。如下所示的 `PyGTKStandardDialog.py` 脚本分别创建了上述标准对话框。

```
# -*- coding:utf-8 -*-
# file: PyGTKStandardDialog.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.labell = gtk.Label('StandardDialog Example')
        self.fixed.put(self.labell, 80, 40)
        self.button1 = gtk.Button('FileChooser')
        self.button1.connect('clicked', self.OnButton1, 'FileChooser')
        self.button2 = gtk.Button('FontChooser')
        self.button2.connect('clicked', self.OnButton2, 'FontChooser')
        self.button3 = gtk.Button('ColorChooser')
        self.button3.connect('clicked', self.OnButton3, 'ColorChooser')

        self.fixed.put(self.button1, 10, 130)
        self.fixed.put(self.button2, 95, 130)
        self.fixed.put(self.button3, 190, 130)
        self.window.add(self.fixed)
        self.labell.show()
        self.button1.show()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
创建标签
添加标签
创建按钮
绑定按钮事件
创建按钮
绑定按钮事件
创建按钮
绑定按钮事件
添加按钮
添加按钮
添加按钮
向窗口中添加 Fixed
显示组件


```

self.button2.show()
self.button3.show()
self.fixed.show()
self.window.show()
def OnButton1(self, widget, data):
    dialog = gtk.FileChooserDialog('Open',
        None,
        gtk.FILE_CHOOSER_ACTION_OPEN,
        (gtk.STOCK_CANCEL,
         gtk.RESPONSE_CANCEL,
         gtk.STOCK_OPEN,
         gtk.RESPONSE_OK))
    filter = gtk.FileFilter()
    filter.set_name('All files')
    filter.add_pattern('*')
    dialog.add_filter(filter)

    filter = gtk.FileFilter()
    filter.set_name('Python')
    filter.add_pattern('*.py')
    filter.add_pattern('*.pyw')
    dialog.add_filter(filter)

    r = dialog.run()
    if r == gtk.RESPONSE_OK:
        print dialog.get_filename()
        dialog.destroy()
def OnButton2(self, widget, data):
    fontdlg = gtk.FontSelectionDialog('Choose Font')
    r = fontdlg.run()
    if r == gtk.RESPONSE_OK:
        print fontdlg.get_font_name()
        fontdlg.destroy()
def OnButton3(self, widget, data):
    colordlg = gtk.ColorSelectionDialog('Choose Color')

    colordlg.colorsels.set_has_palette(True)
    response = colordlg.run()
    if response == gtk.RESPONSE_OK:
        print colordlg.colorsels.get_current_color()
        colordlg.destroy()
def main(self):
    gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

按钮事件处理函数
创建文件打开对话框
设置父窗口
设置对话框标志
添加 Cancel 按钮
Cancel 按钮的返回值
添加 Open 按钮
Open 按钮的返回值
生成 gtk.FileFilter 对象
添加文件类型名
即所有文件
向对话框中添加
gtk.FileFilter 对象
生成 gtk.FileFilter 对象
添加文件类型名
添加文件后缀名
添加文件后缀名
向窗口中添加
gtk.FileFilter 对象
显示对话框

销毁对话框
按钮事件处理函数
创建字体选中对话框
显示对话框

销毁对话框
按钮事件处理函数
创建颜色选择对话框
显示调色板
显示对话框

销毁对话框
定义 main 方法

创建窗口对象

运行 PyGTKStandardDialog.py 脚本后，单击【FileChooser】按钮，将创建如图 14-17 所示的对话框。单击【FontChooser】按钮，将创建如图 14-18 所示的对话框。单击【ColorChooser】按钮，将创建如图 14-19 所示的对话框。

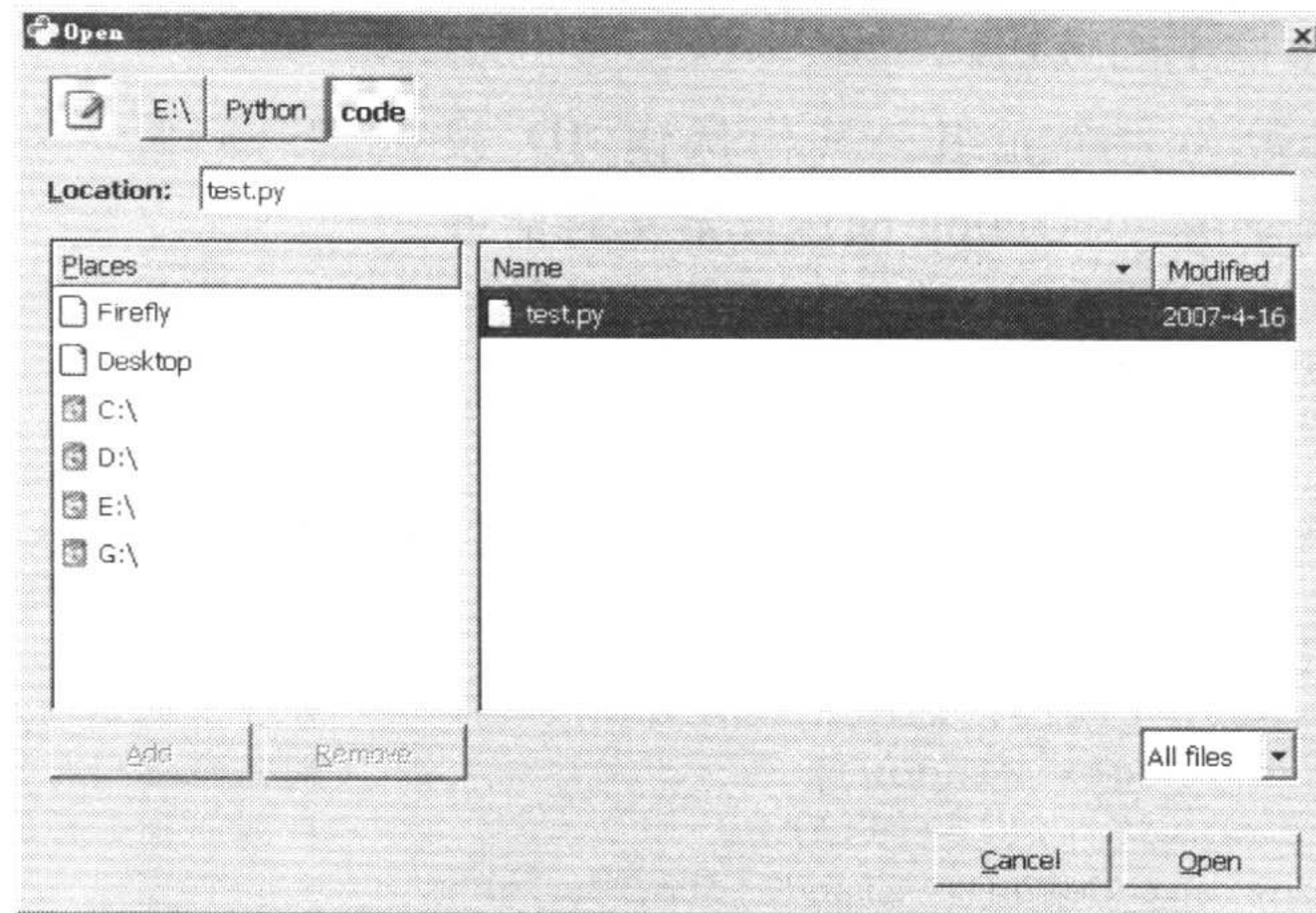


图 14-17 文件打开对话框

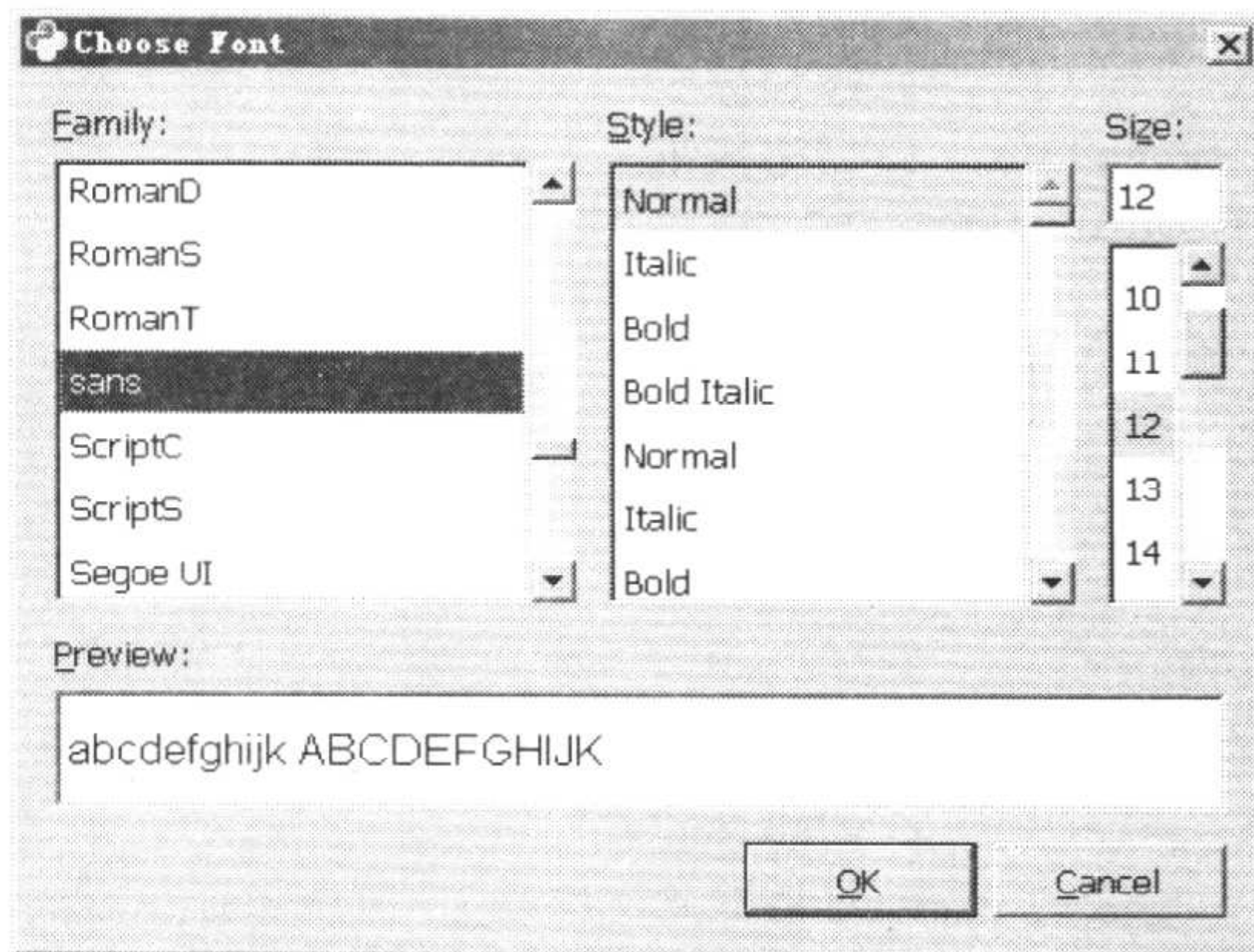


图 14-18 字体选择对话框

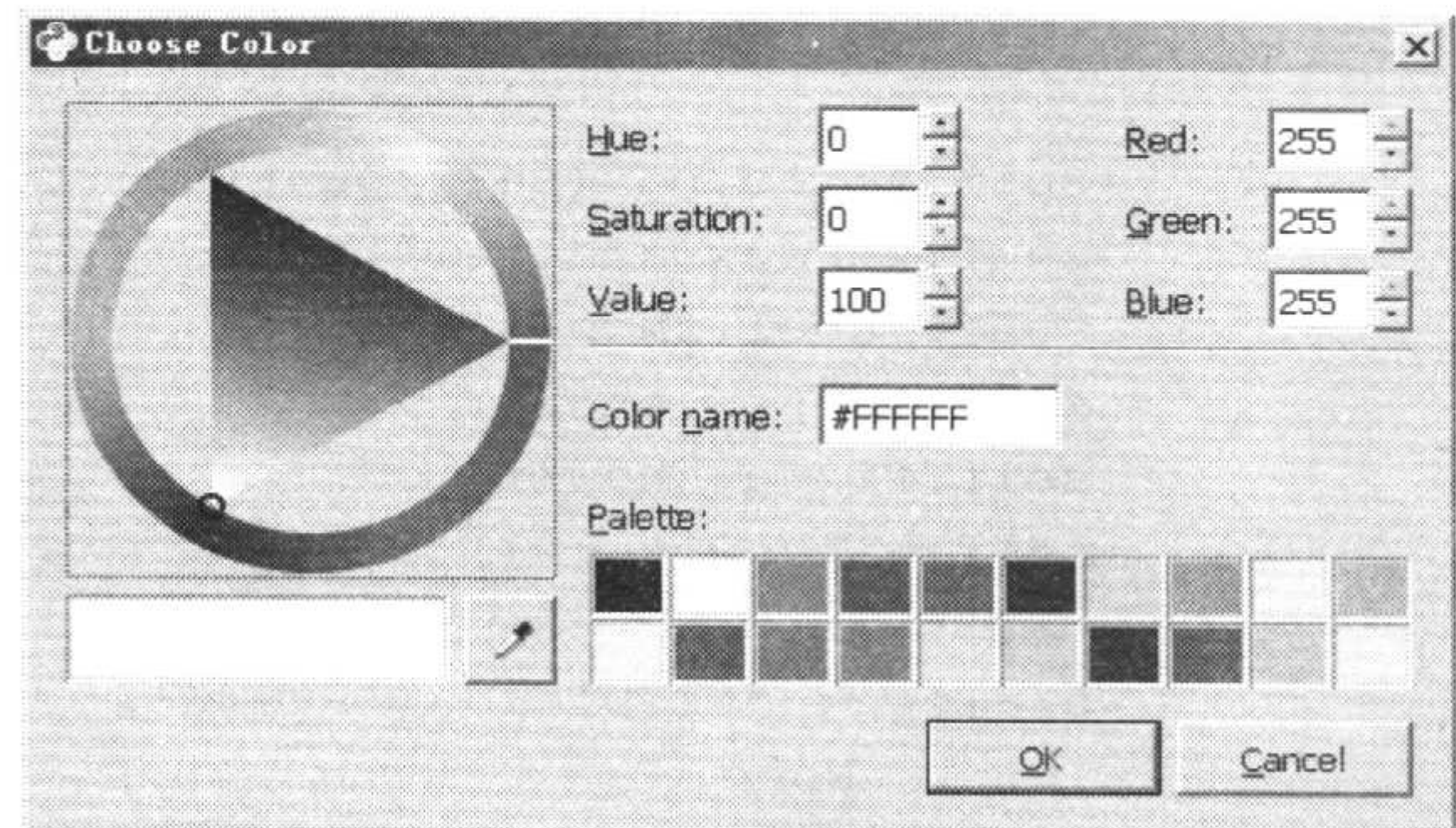


图 14-19 颜色选择对话框

14.3.3 自定义对话框

在 PyGTK 中使用 `gtk.Dialog` 可以创建自定义对话框。所创建的对话框对象和窗口对象一样可以向其中添加组件。由 `gtk.Dialog` 创建的对话框被分为两部分：一部分为 `Vbox` 对象区域，称之为“`vbox`”，可以向其中添加其他的组件；另一部分为 `HBox` 对象区域，称之为“`action_area`”，对话框中的按钮在该区域。`gtk.Dialog` 的原型如下所示。

```
gtk.Dialog(title=None, parent=None, flags=0, buttons=None)
```

其参数含义如下。

- `title`: 对话框的标题。
- `parent`: 对话框的父窗口。
- `flags`: 对话框的创建标志。

- buttons: 对话框中的按钮。

其中 buttons 参数为一个元组，其由按钮 ID 及按钮返回值成对组成。如下所示的 PyGTKDialog.py 脚本使用 gtk.Dialog 创建自定义对话框。

```
# -*- coding:utf-8 -*-
# file: PyGTKDialog.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        self.window = gtk.Window()
        self.window.set_title(title)
        self.window.set_default_size(width, height)
        self.window.connect('destroy', lambda q: gtk.main_quit())
        self.fixed = gtk.Fixed()
        self.label = gtk.Label('Dialog Example')
        self.fixed.put(self.label, 80, 40)
        self.button = gtk.Button('CreateDialog')
        self.button.connect('clicked', self.OnButton, 'CreateDialog')

        self.fixed.put(self.button, 80, 130)
        self.window.add(self.fixed)
        self.label.show()
        self.button.show()
        self.fixed.show()
        self.window.show()

    def OnButton(self, widget, data):
        dialog = gtk.Dialog('PyGTK',
                             None,
                             gtk.DIALOG_MODAL,
                             (gtk.STOCK_CANCEL,
                              gtk.RESPONSE_CANCEL,
                              gtk.STOCK_OK,
                              gtk.RESPONSE_OK))
        fixed = gtk.Fixed()
        dialog.vbox.pack_start(fixed)
        label = gtk.Label('Input')
        fixed.put(label, 10, 5)
        entry = gtk.Entry()
        fixed.put(entry, 50, 5)
        fixed.show()
        label.show()
        entry.show()
        r = dialog.run()
        if r == gtk.RESPONSE_OK:
            print entry.get_text()
        dialog.destroy()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
创建标签
添加标签
创建按钮
绑定按钮事件
添加按钮
向窗口中添加 Fixed
显示组件
按钮事件处理函数
创建对话框
对话框父窗口
对话框标志
向对话框中添加 Cancel 按钮
Cancel 按钮的返回值
向对话框中添加 OK 按钮
OK 按钮的返回值
创建 Fixed 组件
向对话框中的 vbox 添加 Fixed 组件
创建标签
向 Fixed 组件中添加标签
创建文本框
向 Fixed 组件中添加文本框
显示各组件
显示对话框并获取其返回值
如果单击 OK 按钮，则输出文本框中的内容


```

def main(self):                                # 定义 main 方法
    gtk.main()
window = MyWindow('PyGTK', 300, 200)           # 创建窗口对象
window.main()

```

运行 PyGTKDialog.py 脚本后单击 **【CreateDialog】** 按钮，将创建如图 14-20 所示的对话框。

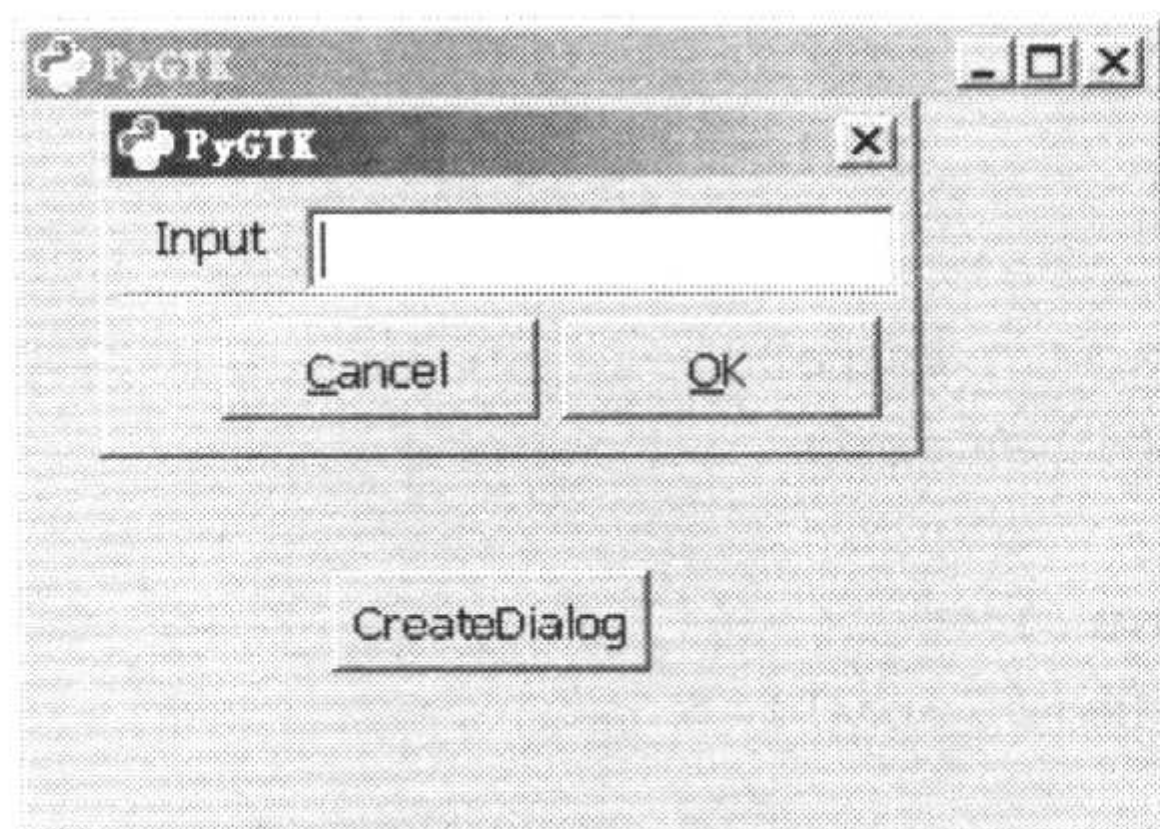


图 14-20 创建自定义对话框

14.4 菜单

在 PyGTK 中提供两种创建菜单的方式，可以一个菜单一个菜单地创建，也可以先定义好菜单项直接创建。在 PyGTK 中菜单的组织方式类似于目录的组件方式，如果预先定义菜单，需要在菜单中使用“/”以表明菜单的位置。

14.4.1 创建菜单

使用 PyGTK 中的 `gtk.MenuBar`、`gtk.Menu`、`gtk.MenuItem` 可以依次创建菜单。如果一次要创建较多的菜单，可以使用 `gtk.ItemFactory`。

1. 依次创建菜单

使用 `gtk.MenuBar` 可以创建一个菜单条，当创建好菜单条后，可以使用其 `append` 方法向其中添加下拉菜单。使用 `gtk.Menu` 可以创建下拉菜单，当创建好下拉菜单后，也可以使用 `append` 方法向其中添加菜单命令。使用 `gtk.MenuItem` 可以创建菜单命令。如下所示的 `PyGTKMenuItem.py` 脚本创建了简单的菜单。

```

# -*- coding:utf-8 -*-
# file: PyGTKMenuItem.py
#
import pygtk                                     # 导入 pygtk 模块
pygtk.require('2.0')                             # 设置 pygtk 所需的 gtk 版本
import gtk                                       # 导入 gtk 模块
class MyWindow():
    def __init__(self, title, width, height):
        window = gtk.Window()                   # 定义窗口类
        window.set_title(title)                 # 定义初始化方法
        window.set_default_size(width, height) # 生成窗口对象
                                                # 设置窗口标题
                                                # 设置窗口大小

```



```

window.connect('destroy', lambda q: gtk.main_quit()) # 关闭窗口退出程序
fixed = gtk.Fixed()                                # 创建 Fixed 组件
window.add(fixed)
filemenu = gtk.Menu()                               # 创建菜单
open = gtk.MenuItem('Open')                         # 创建 Open 菜单命令
open.show()                                         # 显示 Open
close = gtk.MenuItem('Close')                      # 创建 Close 菜单命令
close.show()                                       # 显示 Close
filemenu.append(open)                             # 向菜单中添加 Open
filemenu.append(close)                           # 向菜单中添加 Close
file = gtk.MenuItem('_File')                      # 生成 File 菜单, 下划线表示快捷键
file.set_submenu(filemenu)                       # 向 File 菜单中添加项
file.show()                                       # 显示 File 菜单
editmenu = gtk.Menu()                              # 创建菜单
copy = gtk.MenuItem('Copy')                       # 创建 Copy 菜单命令
copy.show()                                       # 显示 Copy
paste = gtk.MenuItem('Paste')                    # 创建 Paste 菜单命令
paste.show()                                       # 显示 Paste
editmenu.append(copy)                             # 向菜单中添加 Copy
editmenu.append(paste)                           # 向菜单中添加 Paste
edit = gtk.MenuItem('_Edit')                     # 生成 Edit 菜单
edit.set_submenu(editmenu)                       # 向 Edit 菜单中添加项
edit.show()                                       # 显示 Edit 菜单
menubar = gtk.MenuBar()                          # 生成菜单条
menubar.append(file)                             # 向菜单条中添加 File 菜单
menubar.append(edit)                             # 向菜单条中添加 Edit 菜单
fixed.put(menubar, 0, 0)                         # 向 Fixed 组件中添加菜单条
menubar.show()
fixed.show()
window.show()
def main(self):                                   # 定义 main 方法
    gtk.main()
window = MyWindow('PyGTK', 300, 200)              # 创建窗口对象
window.main()

```

运行 PyGTKMenuItem.py 脚本后, 可以看到如图 14-21 所示的【File】菜单和如图 14-22 所示的【Edit】菜单。

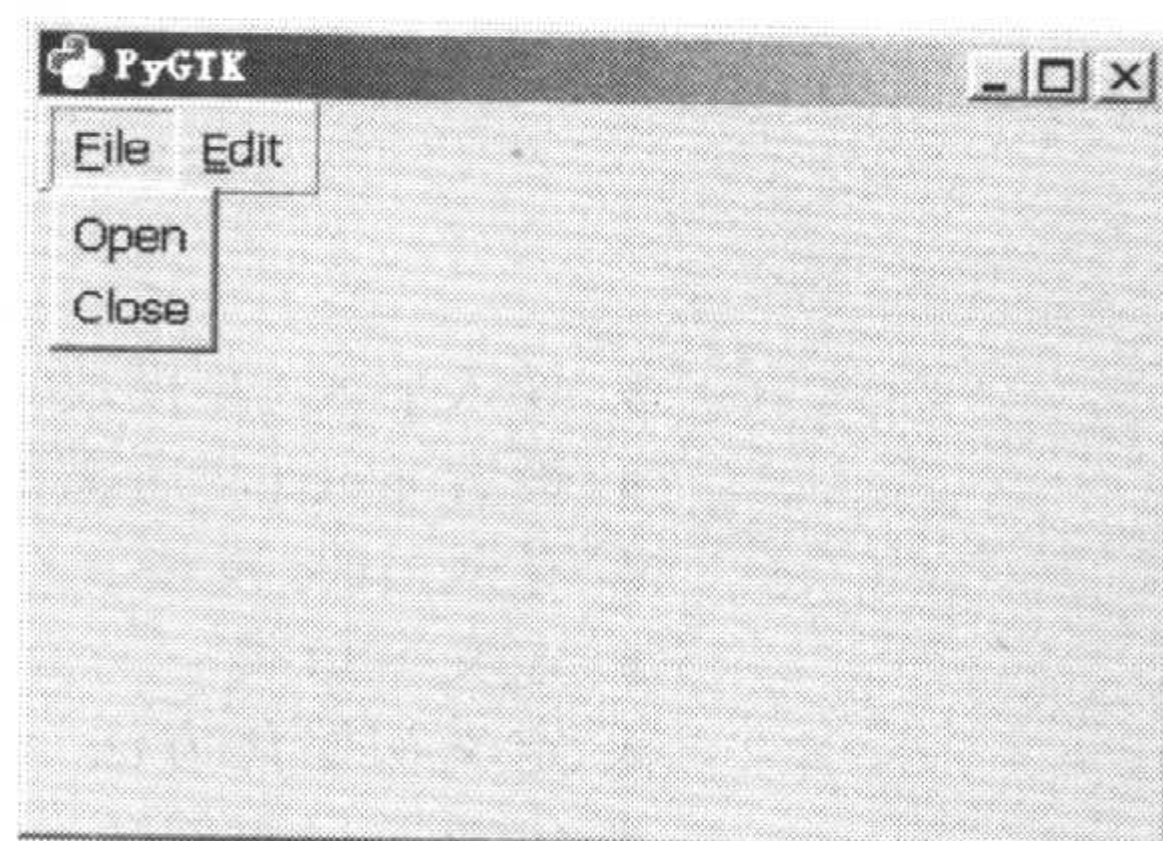


图 14-21 File 菜单

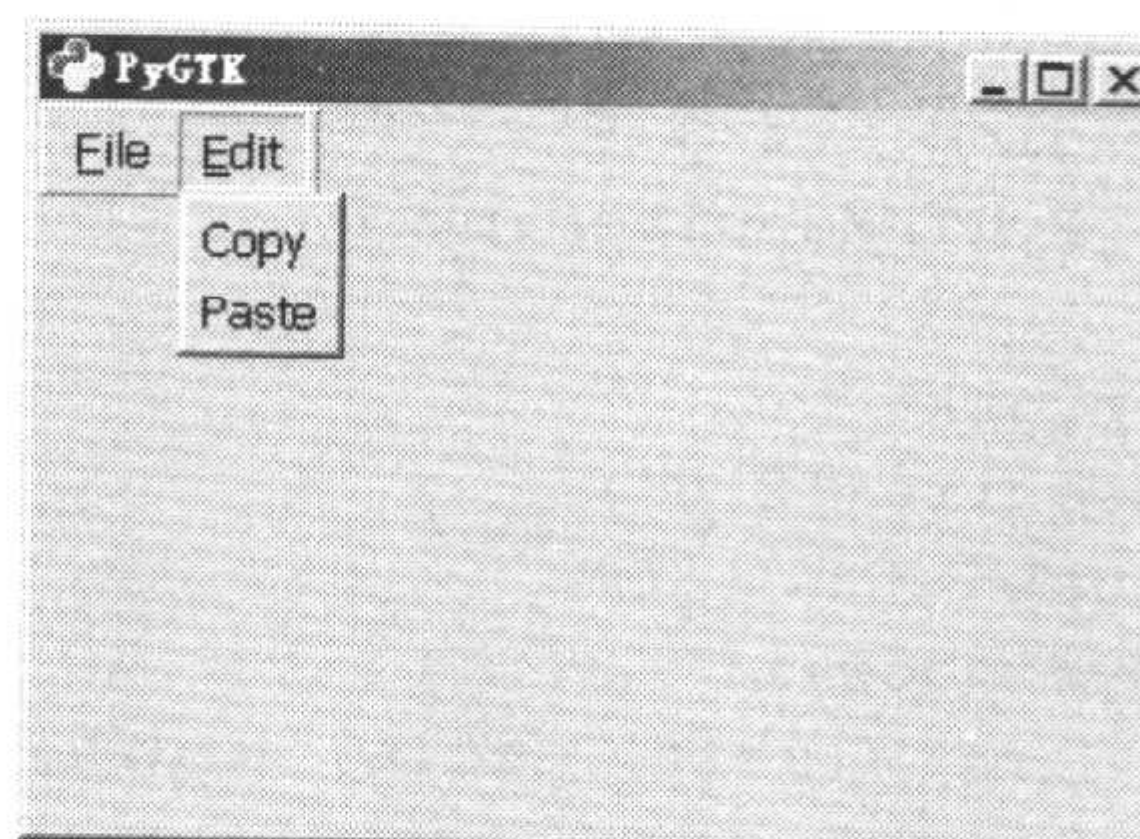


图 14-22 Edit 菜单

2. 使用 gtk.ItemFactory 创建菜单

对于较多的菜单项可以使用 `gtk.ItemFactory` 和 `gtk.MenuBar` 创建。使用 `gtk.ItemFactory` 创建菜单时，首先要用元组的形式定义菜单。元组中的每一项为一个元组，其中包含了菜单的位置、快捷键、菜单命令的回调函数、回调函数动作值（默认为 0）、菜单的样式。如图 14-21 所示的菜单可以写成如下形式的元组。

```
(
    ( '/_File',      None,      None, 0, '<Branch>' ),
    ( '/File/Open',  None,      None, 0, None ),
    ( '/File/Save',  None,      None, 0, None )
)
```

菜单的位置类似于文件的路径，不同的是这里用 Linux 下的 “/” 表示。位于下划线之后的字母表示可以使用 Alt+该字母快捷键访问。如果在菜单项中使用了快捷键，还应该使用 `gtk.AccelGroup` 创建快捷键对象，并将其传递给 `gtk.ItemFactory`。然后使用 window 对象的 `accel_group()` 将快捷键添加到窗口中。

使用 `gtk.ItemFactory` 创建菜单时，需向其传递所创建的菜单类型、菜单路径和快捷键对象。对于菜单路径应为 “<main>”。如下所示的 `PyGTKMenu.py` 创建了一组菜单。

```
# -*- coding:utf-8 -*-
# file: PyGTKMenu.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        window = gtk.Window()
        window.set_title(title)
        window.set_default_size(width, height)
        window.connect('destroy', lambda q: gtk.main_quit())
        fixed = gtk.Fixed()
        window.add(fixed)
        menu_items = (
            ( '/_File',      None,      None, 0, '<Branch>' ),
            ( '/File/Open',  '<control>O', None, 0, None ),
            ( '/File/Save',  '<control>S', None, 0, None ),
            ( '/File/s',     None,      None, 0, '<Separator>' ),
            ( '/File/Close', '<control>Q', None, 0, None ),
            ( '/_Edit',      None,      None, 0, '<Branch>' ),
            ( '/Edit/Copy',   None,      None, 0, None ),
            ( '/Edit/Paste', None,      None, 0, None ),
            ( '/Edit/s',     None,      None, 0, '<Separator>' ),
            ( '/Edit/Cut',   None,      None, 0, None ),
            ( '/_Help',      None,      None, 0, '<Branch>' ),
            ( '/Help/About', None,      None, 0, None )
        )

# 导入 pygtk 模块
# 设置 pygtk 所需的 gtk 版本
# 导入 gtk 模块
# 定义窗口类
# 定义初始化方法
# 生成窗口对象
# 设置窗口标题
# 设置窗口大小
# 关闭窗口，退出程序
# 创建 Fixed 组件

# 菜单
```



```

    )
    accel_group = gtk.AccelGroup()
    itemfactory = gtk.ItemFactory(gtk.MenuBar,
        '<main>', accel_group)
    itemfactory.create_items(menu_items)
    window.add_accel_group(accel_group)
    menubar = gtk.MenuBar()
    menubar = itemfactory.get_widget('<main>')
    fixed.put(menubar, 0, 0)
    menubar.show()
    fixed.show()
    window.show()

def main(self):
    gtk.main()

window = MyWindow('PyGTK', 300, 200)
window.main()

```

创建快捷键对象
创建 ItemFactory 对象

用元组创建菜单
向窗口中添加快捷键
生成菜单条
获得菜单
向 Fixed 组件中添加菜单条
显示各组件

定义 main 方法

创建窗口对象

运行 PyGTKMenu.py 脚本后，单击【File】菜单如图 14-23 所示，单击【Edit】菜单如图 14-24 所示，单击【Help】菜单如图 14-25 所示。

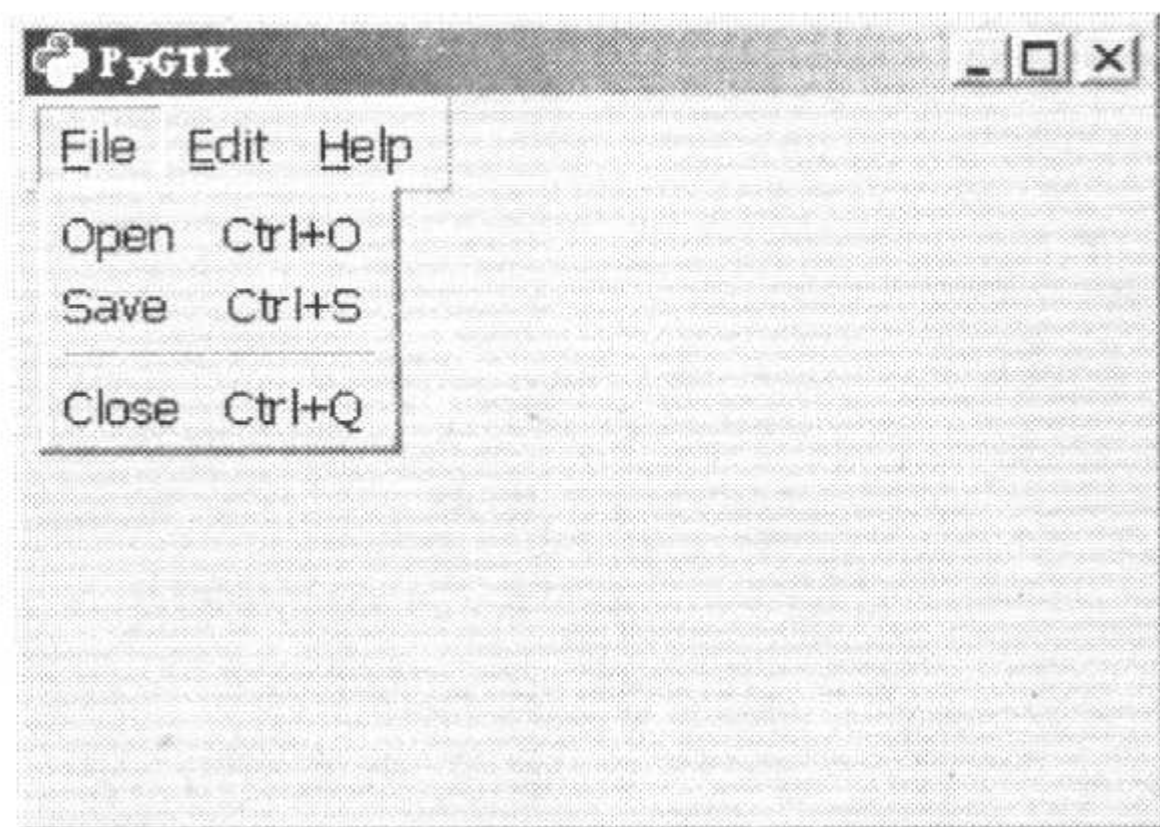


图 14-23 File 菜单

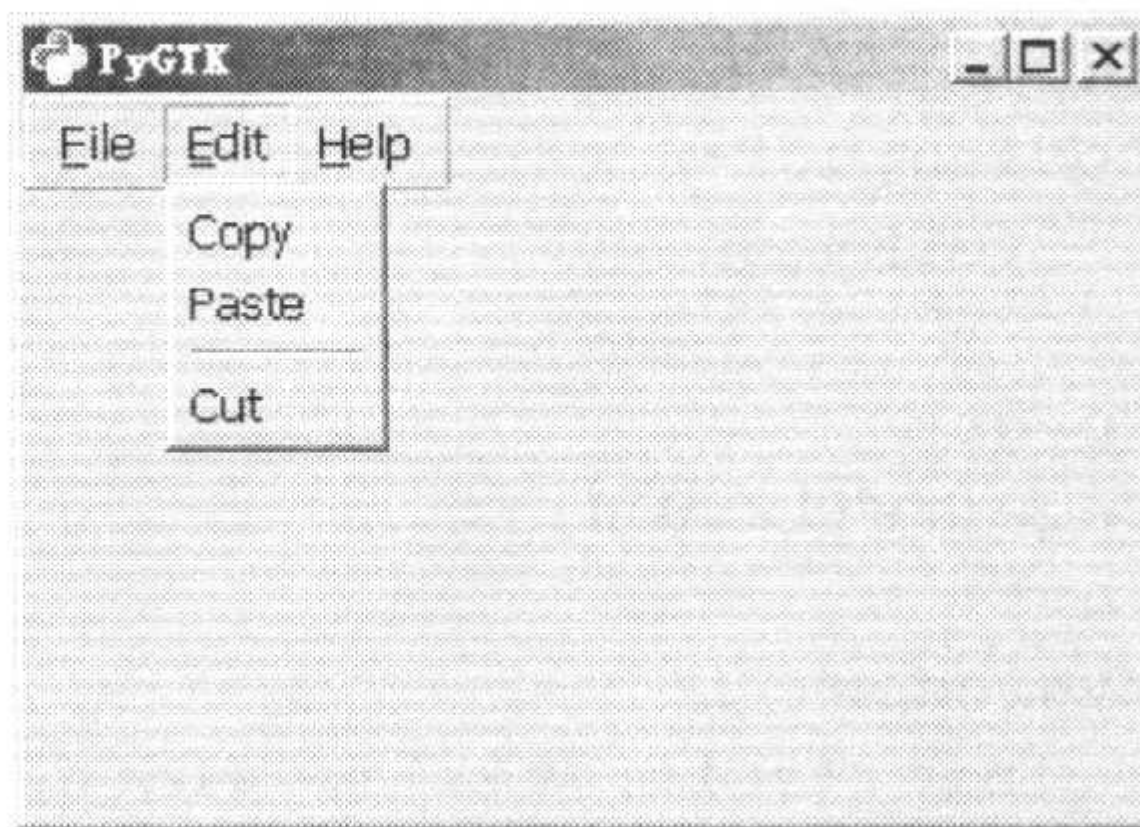


图 14-24 Edit 菜单

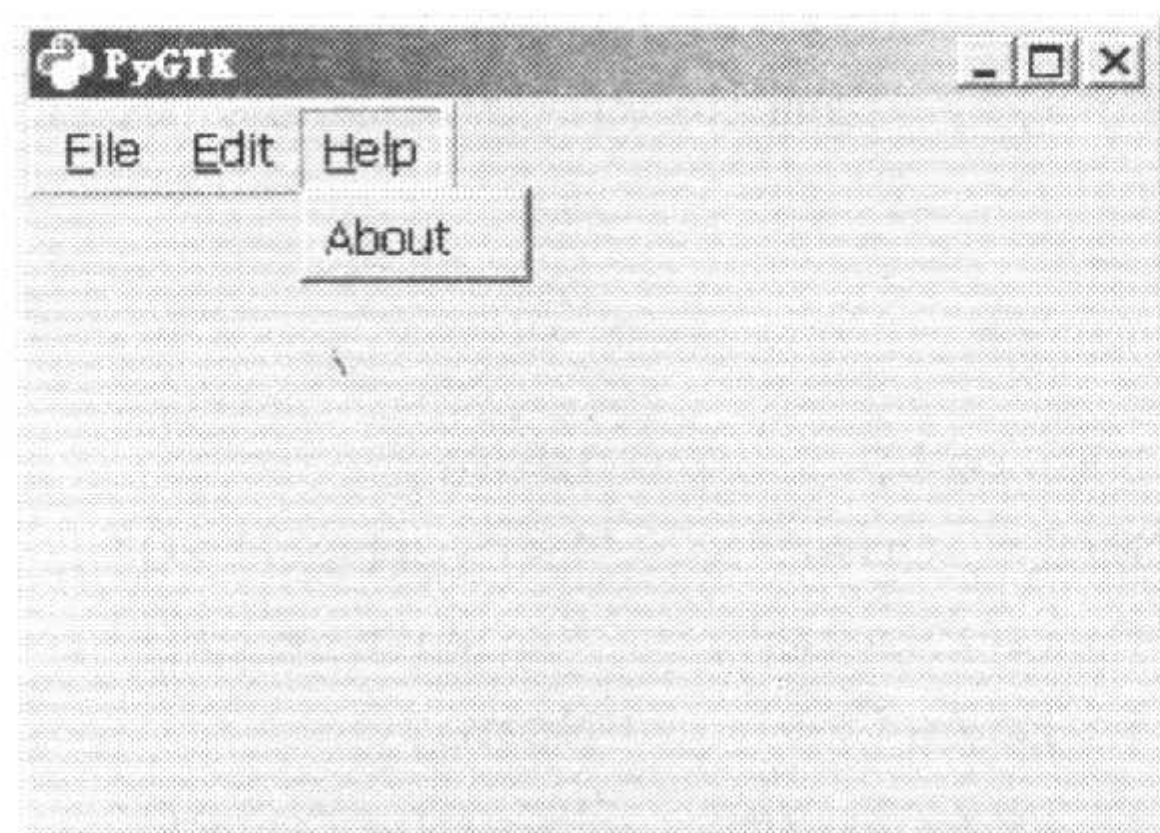


图 14-25 Help 菜单

运行脚本后将出现如下所示的警告。

```

E:\book\code\PyGTKMenu.py:30: DeprecationWarning: use gtk.UIManager
    item_factory = gtk.ItemFactory(gtk.MenuBar, '<main>', accel_group)

```

这是因为菜单项直接写在脚本中而没有使用 XML 格式的文件，出现该警告并不影响程序运行，可以忽略。

14.4.2 菜单事件

菜单事件的绑定同按钮事件的绑定类似。只要使用菜单对象的 `connect` 方法绑定菜单事件的响应函数即可。菜单事件的响应函数也应该按照 `connect` 方法所使用的参数来定义。不同的是菜单事件的事件名为“`activate`”。如下所示的 `PyMenuEvent.py` 脚本使用菜单的 `connect` 方法绑定菜单事件。

```
# -*- coding:utf-8 -*-
# file: PyMenuEvent.py
#
import pygtk
pygtk.require('2.0')
import gtk

class MyWindow():
    def __init__(self, title, width, height):
        window = gtk.Window()
        window.set_title(title)
        window.set_default_size(width, height)
        window.connect('destroy', lambda q: gtk.main_quit())
        fixed = gtk.Fixed()
        window.add(fixed)
        filemenu = gtk.Menu()
        open = gtk.MenuItem('Open')
        open.show()
        open.connect('activate', self.OnOpen, 'Open')
        close = gtk.MenuItem('Close')
        close.connect('activate', self.OnClose, 'Close')
        close.show()
        filemenu.append(open)
        filemenu.append(close)
        file = gtk.MenuItem('_File')
        file.set_submenu(filemenu)
        file.show()
        menubar = gtk.MenuBar()
        menubar.append(file)
        fixed.put(menubar, 0, 0)
        menubar.show()
        fixed.show()
        window.show()

    def OnOpen(self, widget, data):
        dialog = gtk.FileChooserDialog('Open',
                                       None,
                                       gtk.FILE_CHOOSER_ACTION_OPEN,
                                       (gtk.STOCK_CANCEL, gtk.RESPONSE_CANCEL,
                                        gtk.STOCK_OPEN, gtk.RESPONSE_OK))
        dialog.set_default_response(gtk.RESPONSE_OK)
        response = dialog.run()
        dialog.destroy()
```

导入 pygtk 模块
设置 pygtk 所需的 gtk 版本
导入 gtk 模块
定义窗口类
定义初始化方法
生成窗口对象
设置窗口标题
设置窗口大小
关闭窗口，退出程序
创建 Fixed 组件

创建菜单
创建 Open 菜单命令
显示 Open
绑定菜单事件
创建 Close 菜单命令
绑定菜单事件
显示 Close
向菜单中添加 Open
向菜单中添加 Close
生成 File 菜单，下划线表示快捷键
向 File 菜单中添加项
显示 File 菜单
生成菜单条
向菜单条中添加 File 菜单
向 Fixed 组件中添加菜单条
显示各组件

处理菜单事件
创建打开文件对话框


```

def OnClose(self, widget, data):
    gtk.main_quit()
def main(self):
    gtk.main()
window = MyWindow('PyGTK', 300, 200)
window.main()

```

处理菜单事件
退出程序
定义 main 方法
创建窗口对象

14.5 资源文件

在 PyGTK 中可以使用两种形式的资源文件。一种是文本形式的，类似于 Windows 下资源文件的格式。另一种则是 XML 形式，类似于 wxPython 中的资源文件格式。由于 XML 格式的资源文件可以使用 Glade 创建，使用十分简便。因此，本节以 XML 形式的资源为例进行讲解。

14.5.1 使用 Glade 创建资源文件

Glade 是可视化界面设计工具，它用于生成 GTK 的界面代码。Glade 生成的资源文件同样可以被 PyGTK 使用。使用 Glade 可以简化程序，而且 Glade 功能十分强大，在脚本中使用 gtk.glade 模块可以十分方便地创建 GUI 界面。

1. 安装 Glade

Glade 是开源软件，可以从其官方网站 <http://glade.gnome.org> 下载源代码进行编译安装。对于 Windows 用户而言，可以到 <http://gladewin32.sourceforge.net> 下载编译好 Windows 版本的 Glade。

由于已经安装了 PyGTK，因此只需下载 Glade，而不必下载 GTK+ 的运行库。如果没有安装 PyGTK 而又需要使用 Glade 时，则需要下载 GTK+ 程序的运行库，才可以使用 Glade。

<http://gladewin32.sourceforge.net> 提供的 Windows 版的 Glade 为压缩包。以 glade-3.0.2-win32-1.zip 为例，下载完成将其解压到某一目录，例如“C:\Glade”。进入“C:\Glade\bin”目录，运行其中的 glade-3.exe，如图 14-26 所示。

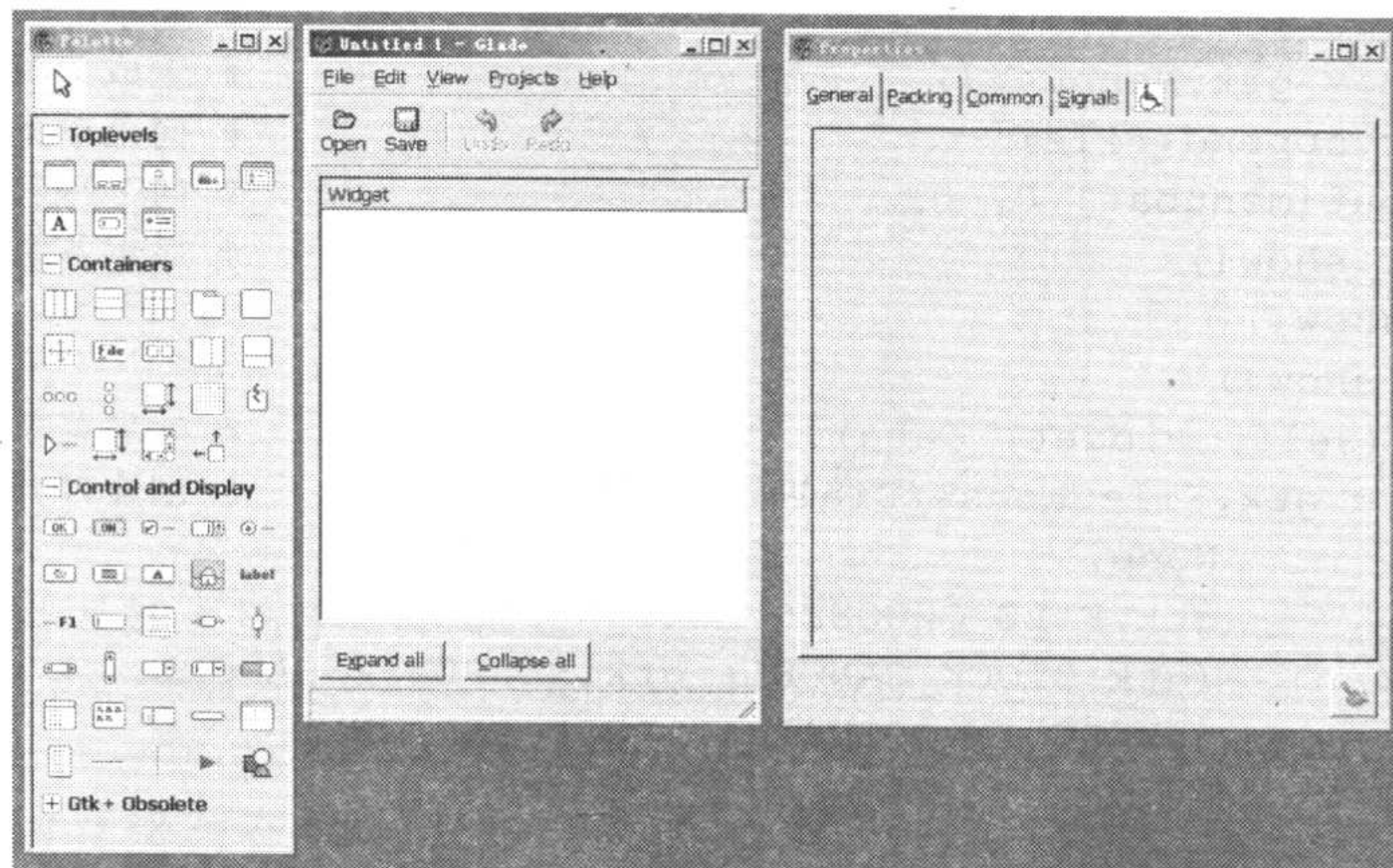


图 14-26 Glade 窗口

可以看到与一般的 Windows 应用程序不同, Glade 由 3 个窗口组成: 一个工程管理窗口 (位于中间的窗口)、一个常用组件窗口 (位于左侧的窗口) 和一个属性管理窗口 (位于右侧的窗口)。

2. 创建资源文件

Glade 的使用与 VC++ 6.0 中的资源编辑器类似, 创建一个资源文件的过程如下所示。

(1) 单击 **【File】|【New】** 新建一个工程。

(2) 单击常用组件窗口中的  按钮, 新建一个窗口, 如图 14-27 所示。

(3) 在属性窗口中的 **【Name】** 文本框中输入窗口名 “window”, 在脚本中即可使用 “window” 载入所创建的资源。在 **【Window Title】** 中输入窗口的标题 “PyGTK Glade”, 如图 14-28 所示。

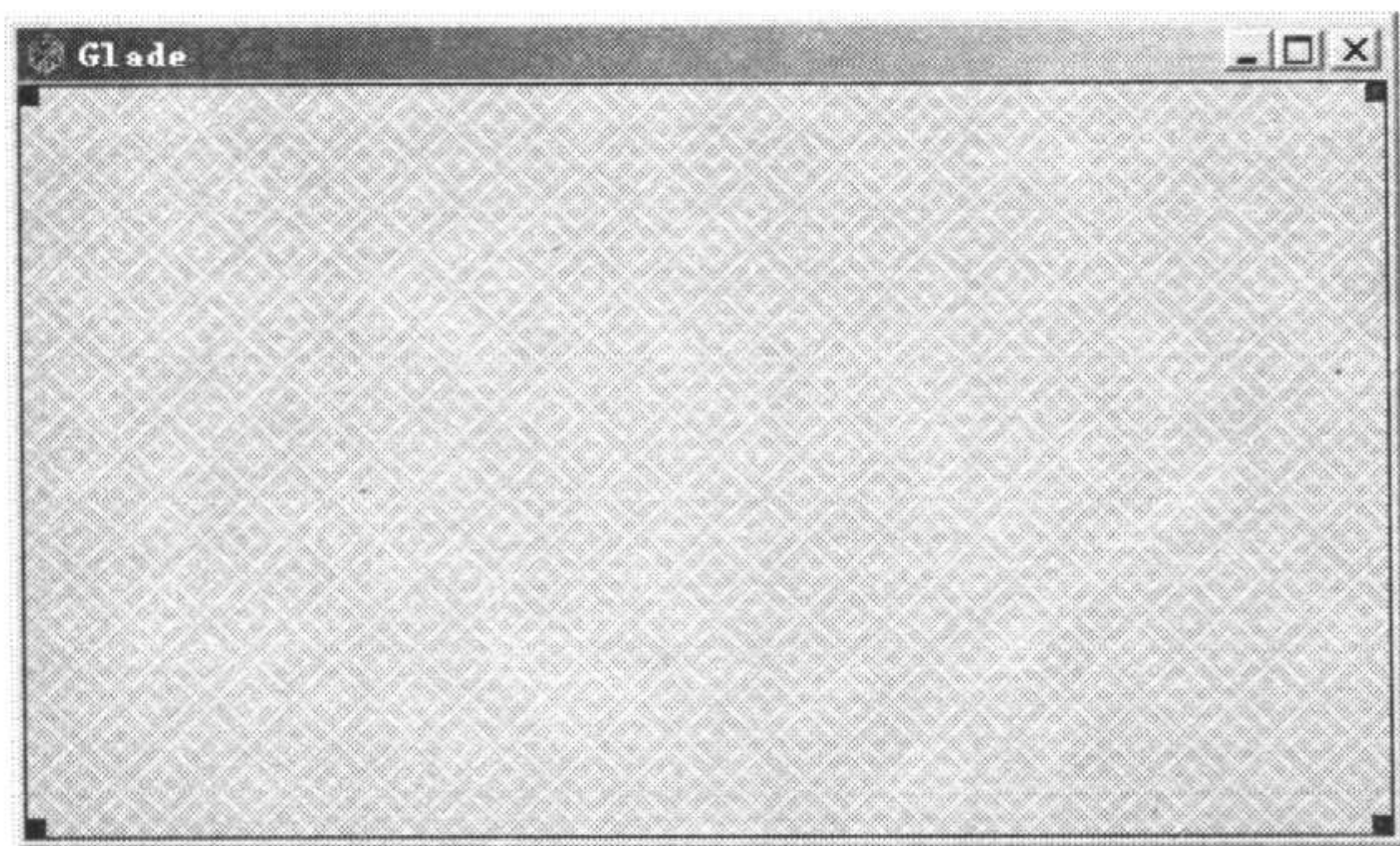


图 14-27 创建的 window 窗口

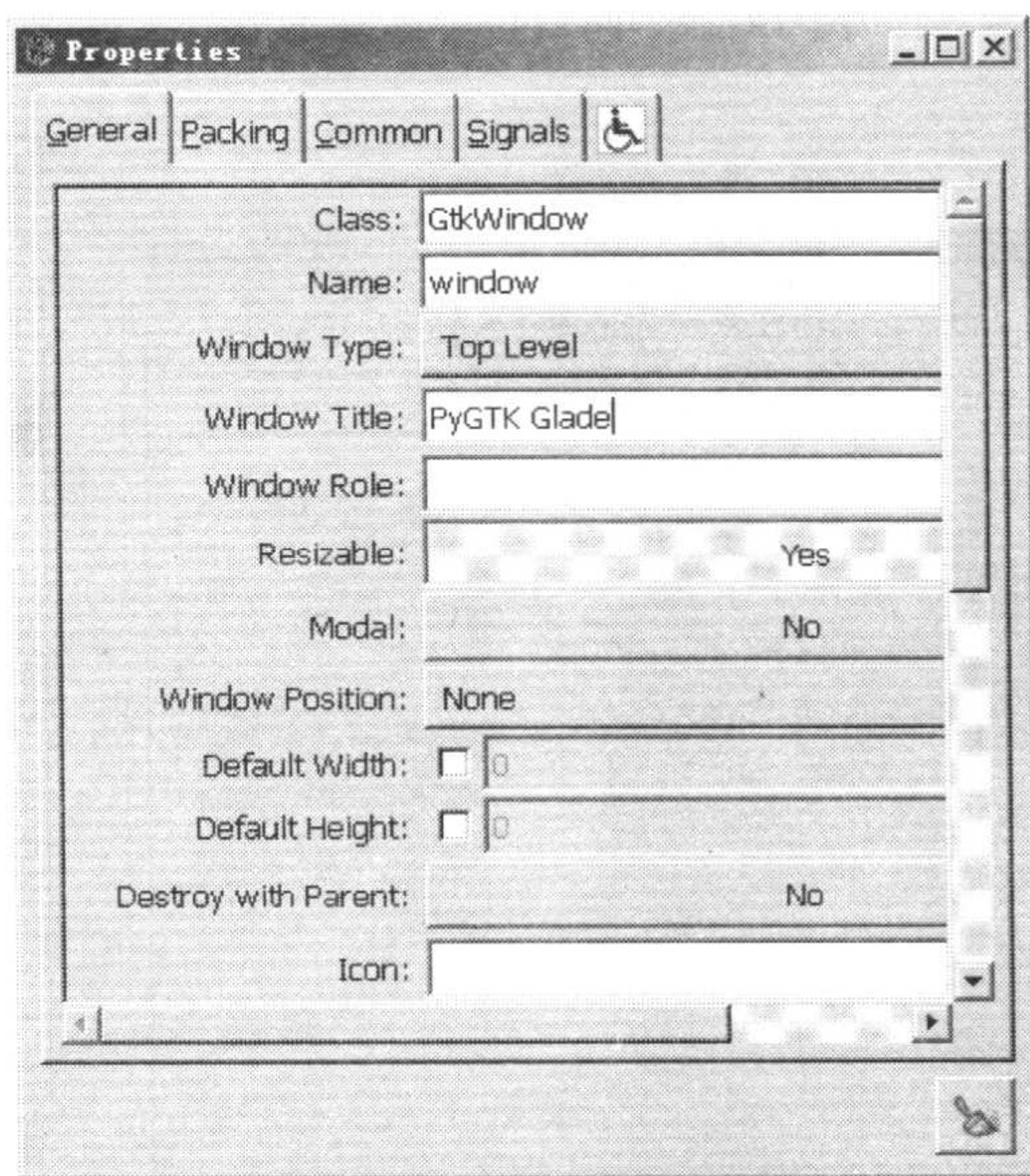





图 14-28 window 窗口的属性窗口

(4) 单击常用组件窗口中的  按钮, 然后单击如图 14-27 所示的窗口, 向窗口中添加 Fixed 组件。

(5) 单击常用组件窗口中的  按钮, 然后单击如图 14-27 所示的窗口, 向窗口中添加一个标准按钮。重新调整菜单的位置和大小, 如图 14-29 所示。

(6) 单击常用组件窗口中的  按钮, 然后单击如图 14-29 所示的窗口, 向窗口中添加 TextView 组件。重新调整其大小, 如图 14-30 所示。

(7) 双击如图 14-30 所示窗口中的 **【File】** 菜单, 选中弹出菜单中的 **【Quit】** 命令。此时属性窗口如图 14-31 所示。单击属性窗口中的 **【Signals】** 标签, 选中 **【GtkMenuItem】|【activate】**

项。在【Handler】一列中填入“OnQuit”，在【User data】一列中填入“Quit”，绑定【Quit】菜单事件，如图 14-32 所示。

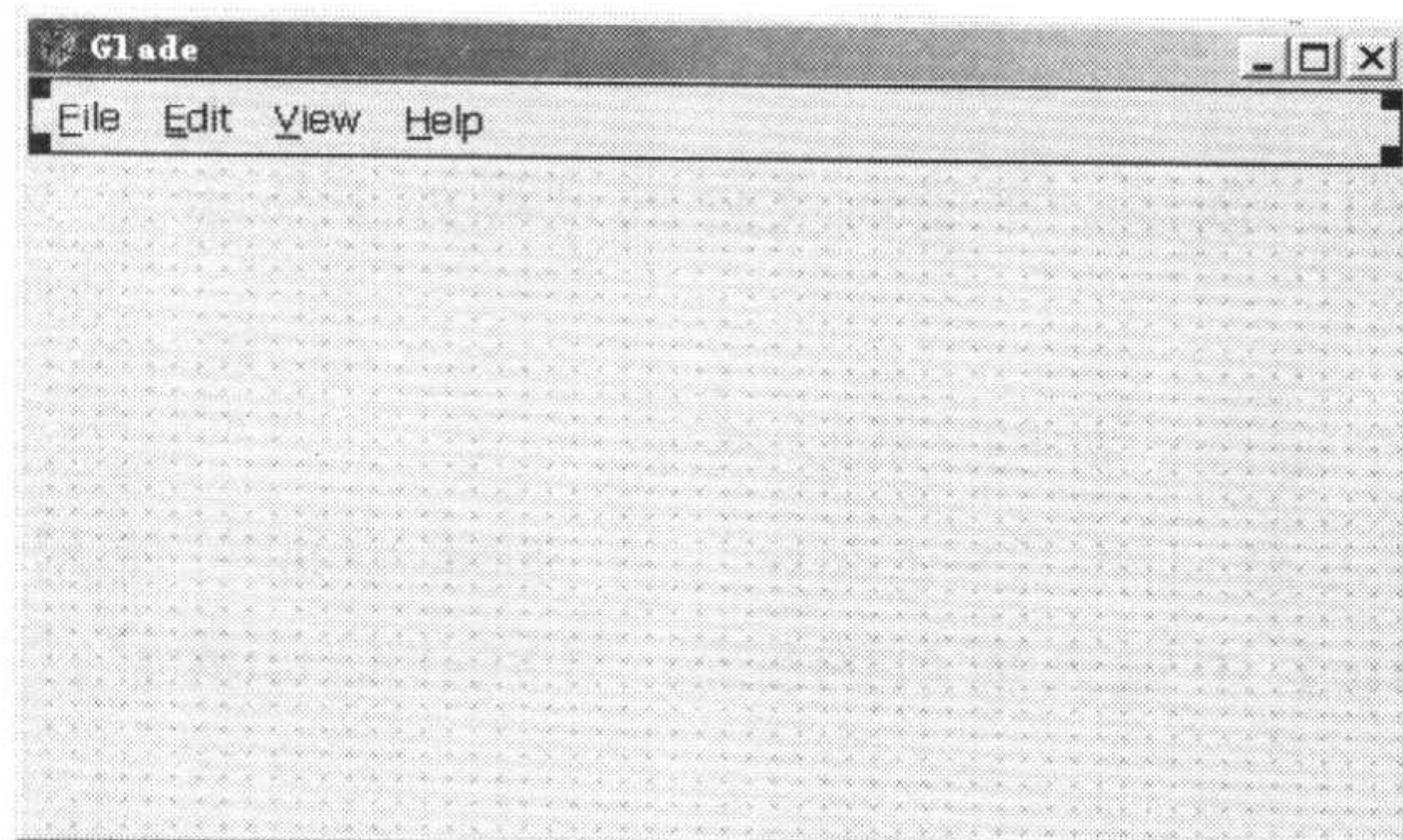


图 14-29 向窗口中添加菜单

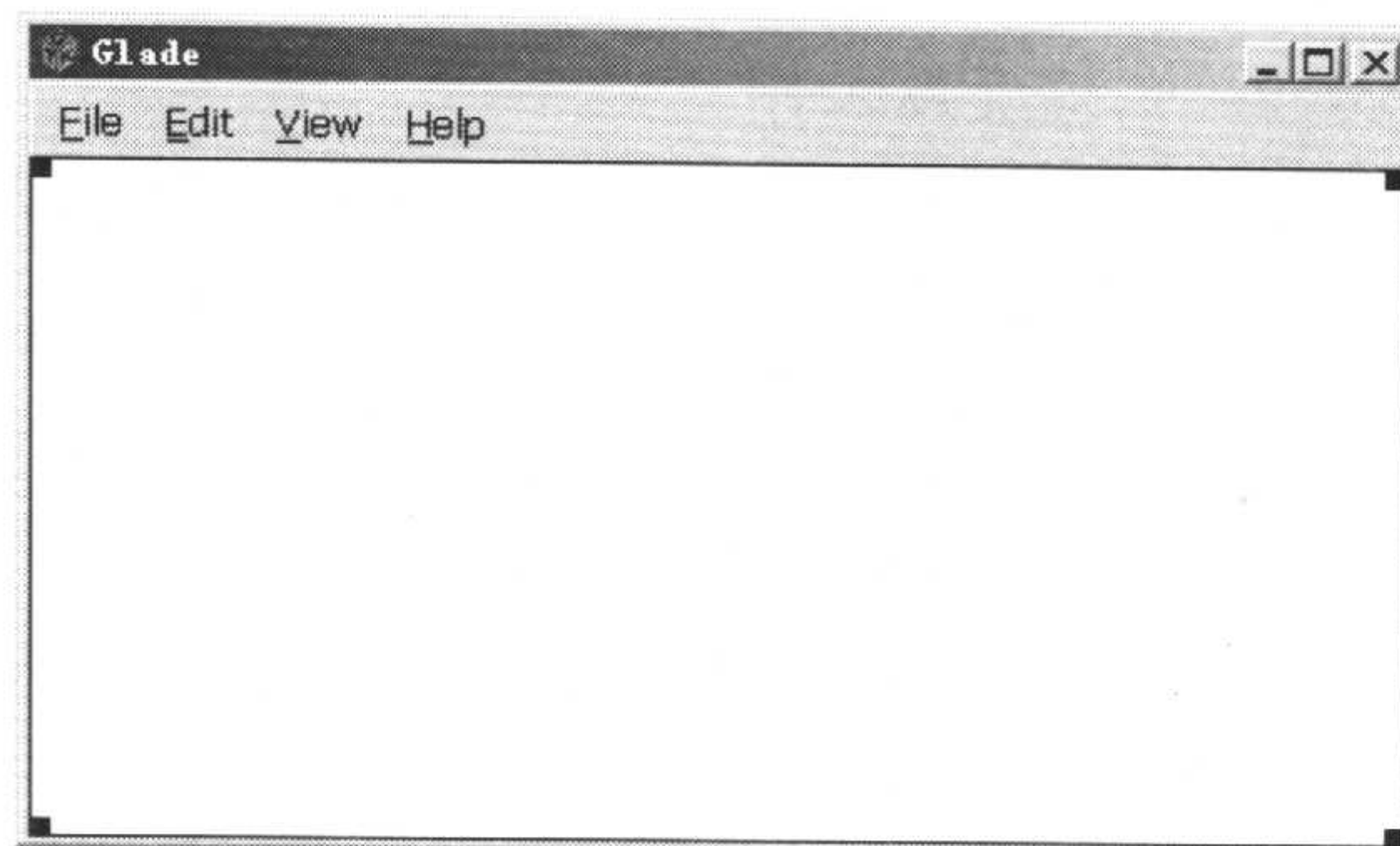


图 14-30 向窗口中添加多行文本框

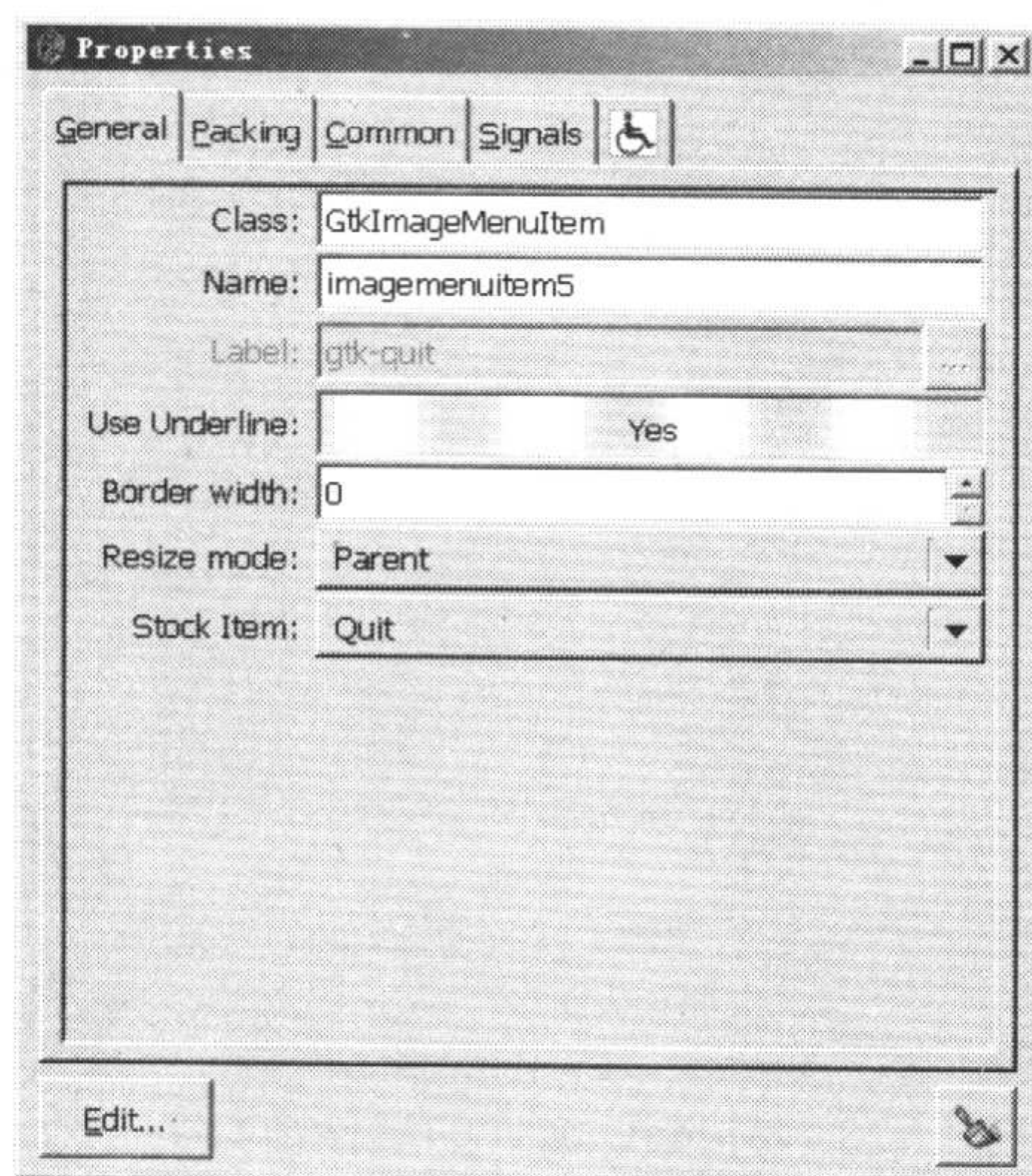


图 14-31 Quit 菜单命令的属性窗口

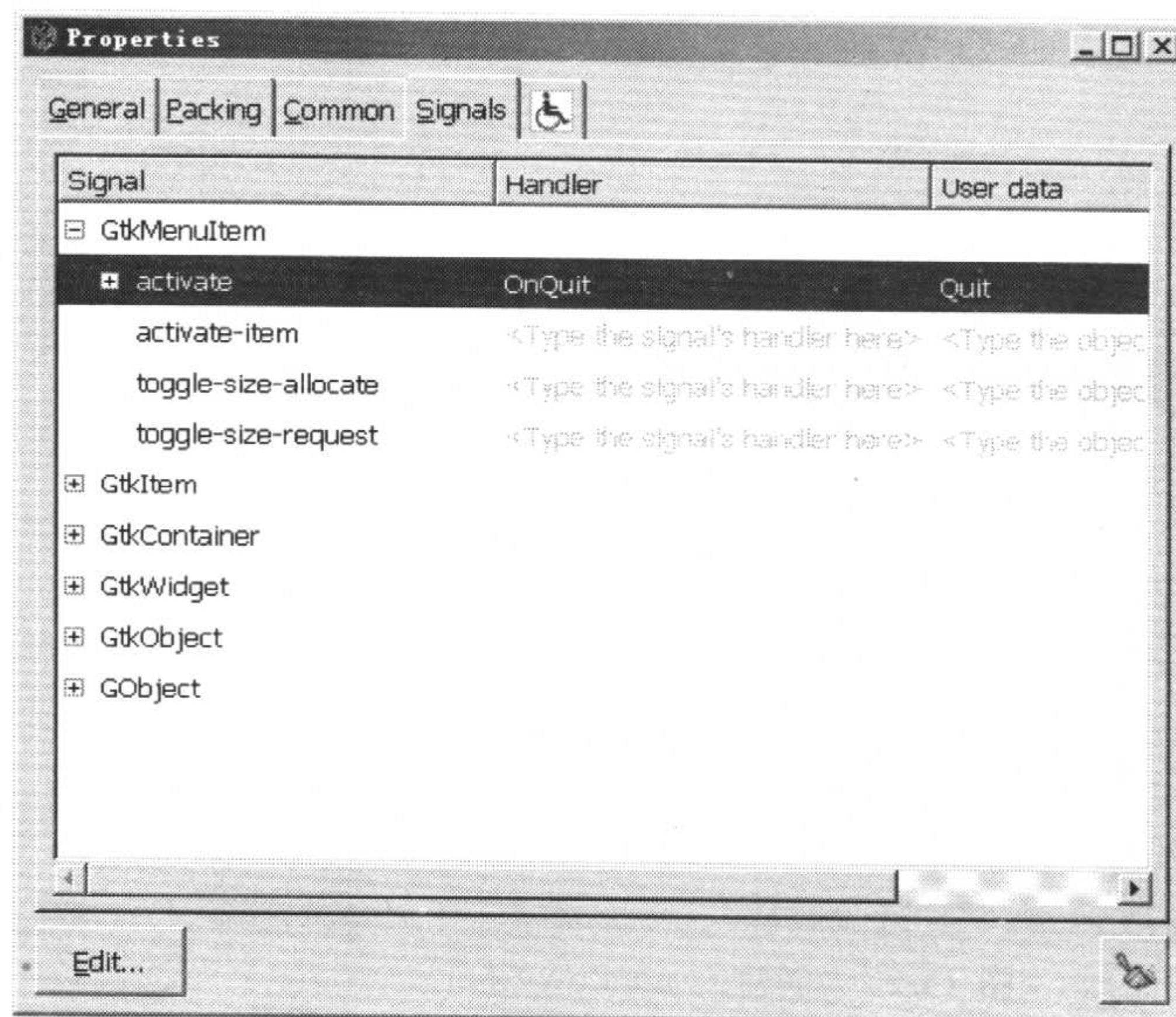


图 14-32 绑定菜单事件

(8) 单击菜单【File】|【Save】命令，将工程保存为“res.glade”

14.5.2 使用资源文件

在脚本中使用 Glade 创建的资源文件只需载入资源文件。如果在资源文件中添加了事件信号，在脚本中也应该载入。在脚本中使用资源文件主要使用 `gtk.glade.XML` 类的以下几个方法。

- `get_widget()`: 获得资源文件中的组件。

- signal_connect(): 绑定信号事件。
- signal_autoconnect(): 绑定多个信号事件。

对于 signal_autoconnect() 方法，其参数为一个字典。以上一节所添加的【Quit】命令的信号为例，将其写在参数字典中，关键字为“OnQuit”，值为事件处理函数。如下所示的 PyGTKGlade.py 脚本使用 gtk.glade 模块载入资源文件。

```
# -*- coding:utf-8 -*-
# file: PyGTKGlade.py
#
import pygtk
pygtk.require('2.0')
import gtk
import gtk.glade
class MyWindow():
    def __init__(self):
        res = gtk.glade.XML('res.glade')

        window = res.get_widget('window')
        signal = { 'OnQuit' : self.OnQuit }
        res.signal_autoconnect(signal)
        window.connect('destroy', lambda q:gtk.main_quit())
        window.show()
    def OnQuit(self, widget):
        gtk.main_quit()
    def main(self):
        gtk.main()
window = MyWindow()
window.main()
```

定义窗口类
定义初始化方法
载入资源文件，生成
gtk.glade.XML 实例
载入 window
创建信号字典
绑定信号事件
窗口关闭，则退出程序
Quit 菜单命令处理事件
定义 main 方法
调用 gtk.main 方法
创建窗口对象

运行 PyGTKGlade.py 脚本后将创建如图 14-33 所示的窗口。

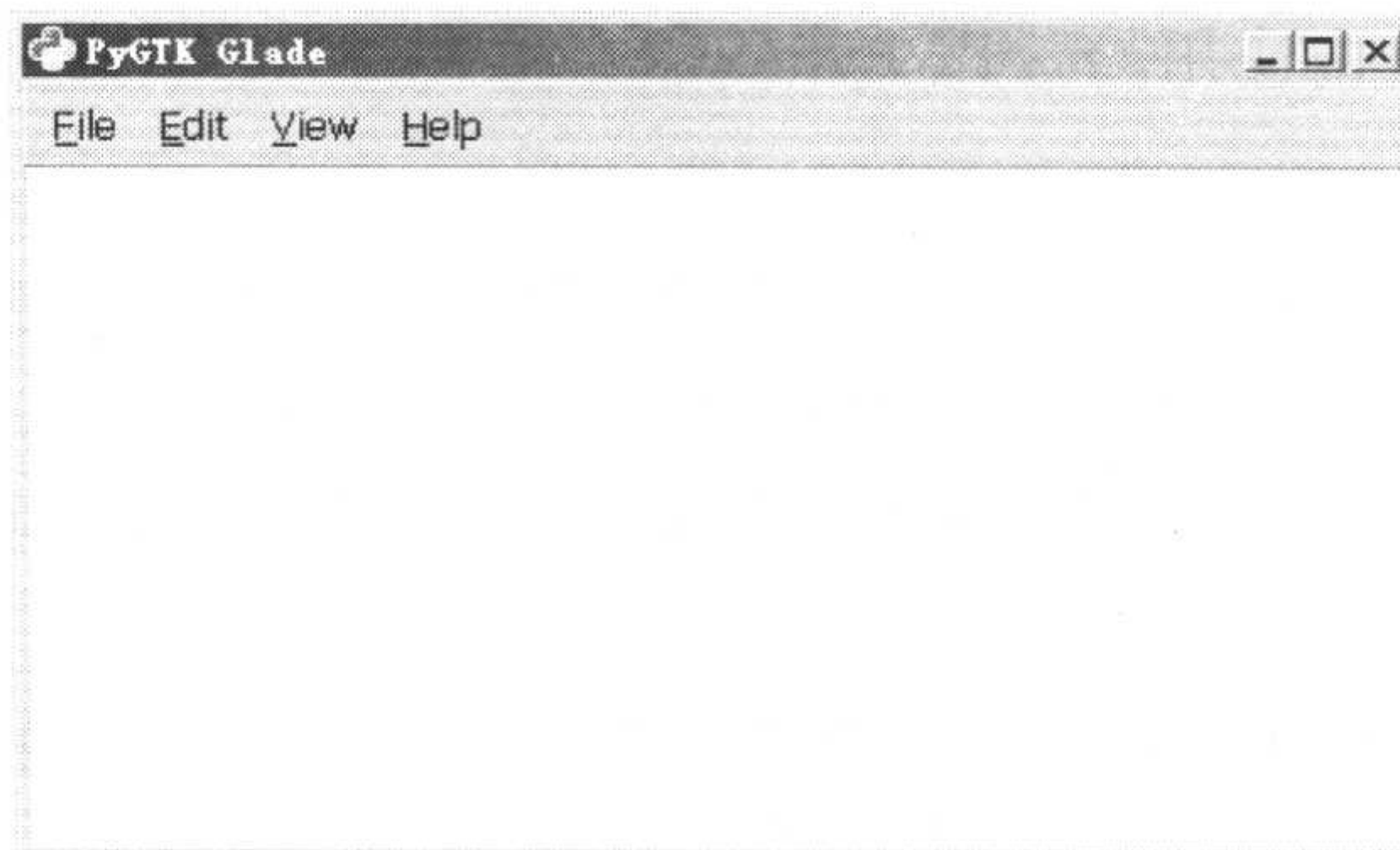


图 14-33 使用 Glade 资源文件创建窗口

第 15 章 使用 PyQT 编写 GUI

PyQt 是对 Qt 的封装。Qt 是面向对象的图形用户界面库，可以在多个操作系统上使用。与其他的开源 GUI 库相比，Qt 显得过于庞大。另外，Qt 虽然是开源 GUI 库，但其许可证限制较复杂。针对不同的系统有不同的限制。如果要使用 Qt 编写商业软件，则需购买许可。PyQt 也严格遵循了 Qt 的许可证。

15.1 PyQt 概述

Qt 是比较早的图形用户界面库，但 Qt 具有较为严格的许可证限制。PyQt 具有和 Qt 一样的许可证，在 Windows 下使用 PyQt 较复杂。对于非商业用途，可以免费使用 PyQt。

15.1.1 PyQt 的安装

PyQt 的安装较为复杂。因为许可证的限制，PyQt 的安装包中并不包含 Qt，因此在安装 PyQt 之前应首先安装 Qt。在 Windows 下 Qt 的安装也较复杂，在 Windows 下 Qt 需要 MingW 的支持。在 Windows 下使用 Qt 最好使用 Qt 4，因为 Qt 4 更改了 Windows 下的许可证，对于非商业目的的应用可以自由使用 Qt。以 Python 2.5 和 PyQt 4.2.3 为例，在 Windows 下其安装过程如下所示。

(1) 到 Qt 的官方网站 <http://www.trolltech.com/developer/downloads/qt/windows> 下载 Windows 下 Qt 4 的安装程序 qt-win-opensource-4.2.3-mingw.exe。

(2) 运行 qt-win-opensource-4.2.3-mingw.exe，如图 15-1 所示。单击【Next】按钮进入下一步安装。

(3) 直接单击【Next】按钮，进入安装协议界面，选中【I accept the terms in the License Agreement】单选框，单击【Next】按钮，进入下一步安装，如图 15-2 所示。

(4) 直接单击【Next】按钮，按照默认安装进入安装路径选择界面，根据需要设置安装路径，如图 15-3 所示。

(5) 单击【Next】按钮进入下一步安装，按照默认设置直接单击【Next】按钮，进入 MinGW 路径选择界面。如果未安装 MinGW，则可以按图 15-4 所示设置自动下载安装

第15章 使用PyQT编写GUI

MinGW。如果已经安装 MinGW，则可以直接选取 MinGW 的安装路径。



图 15-1 Qt 安装程序

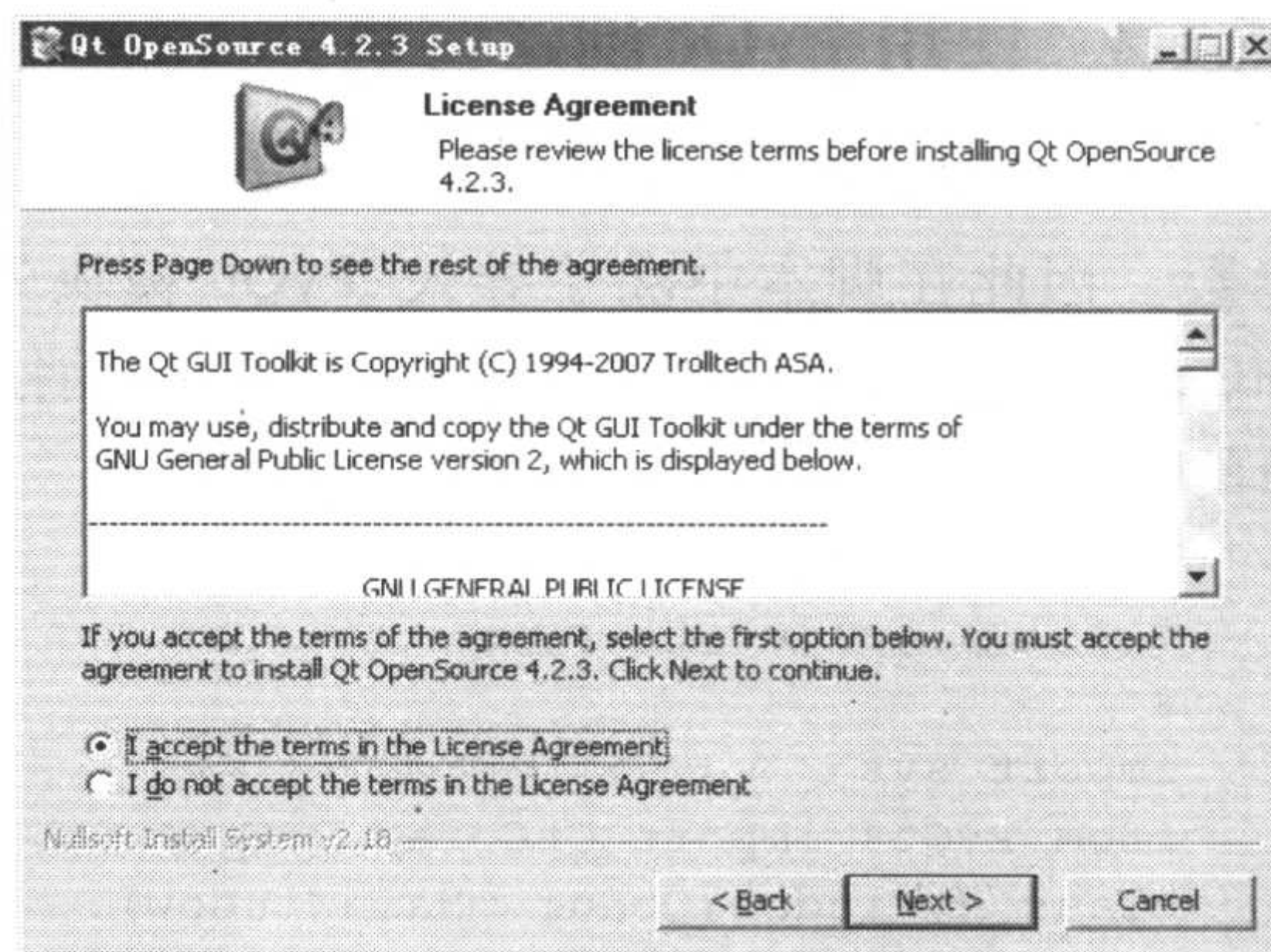


图 15-2 安装协议

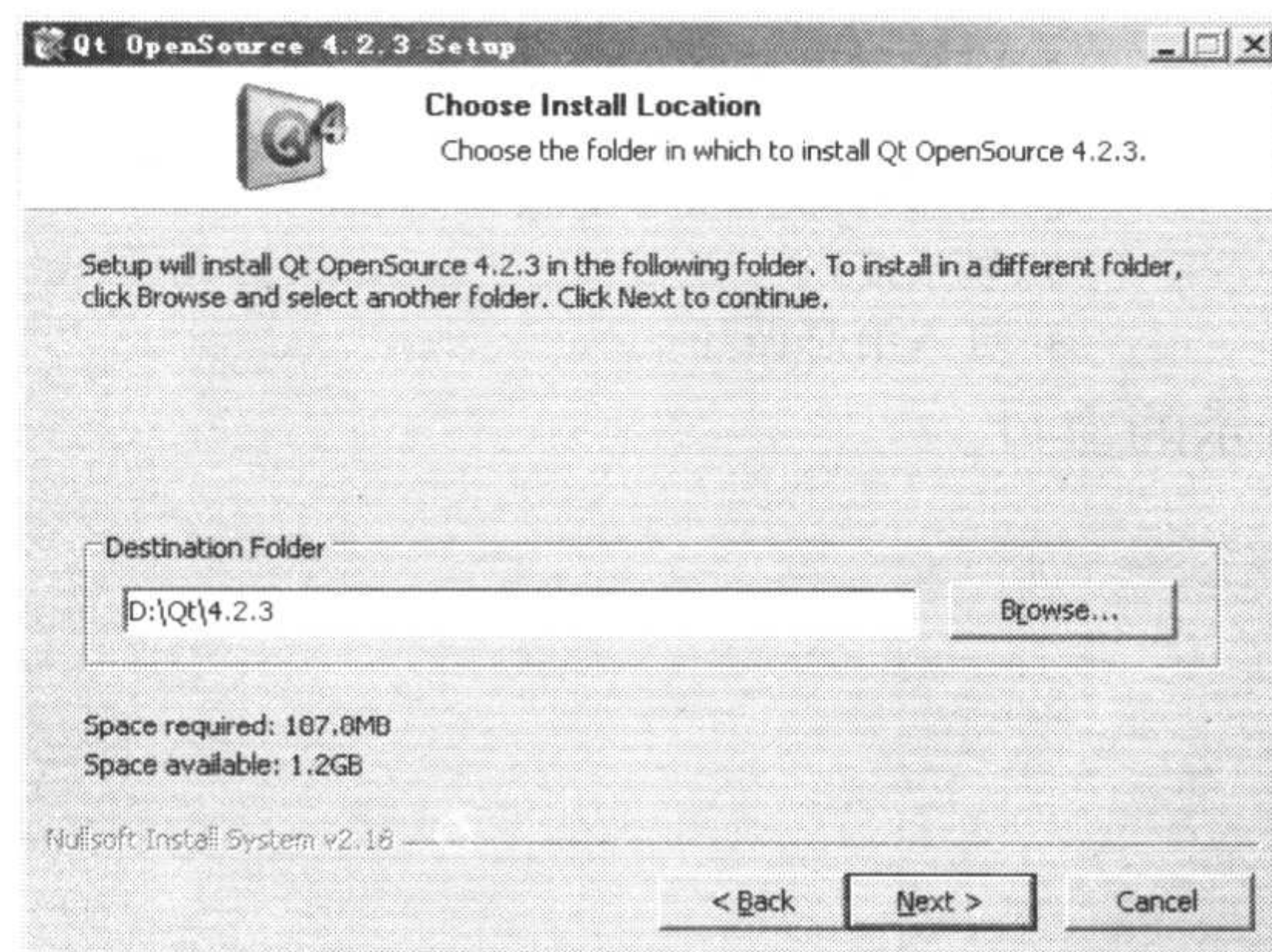


图 15-3 选择安装路径

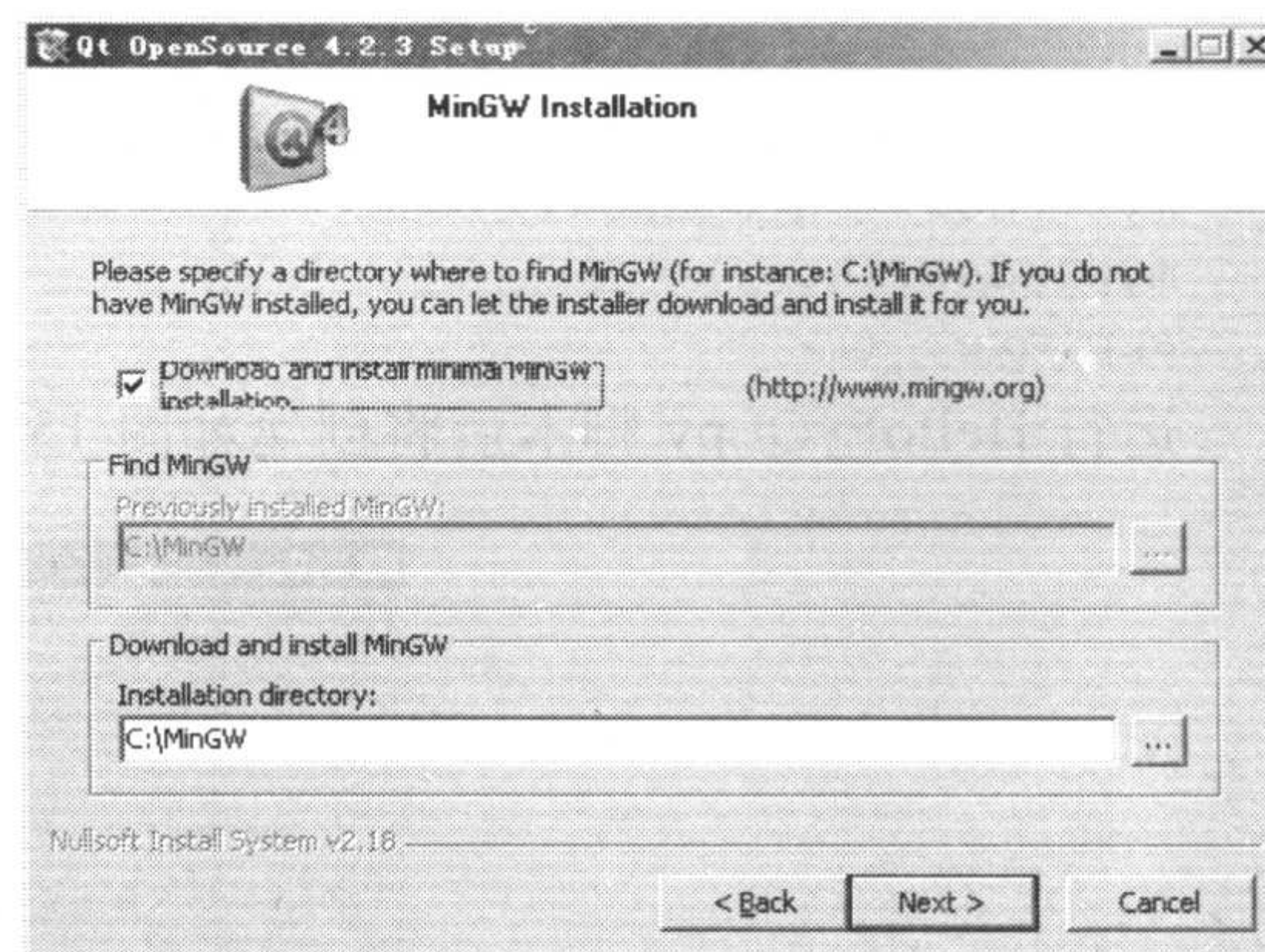


图 15-4 安装 MinGW

(6) 剩下的步骤只需单击【Next】按钮，按照默认设置即可完成 Qt 的安装。

(7) 到 PyQt 的官方网站 <http://www.riverbankcomputing.co.uk/pyqt/index.php> 下载 PyQt 的 Windows 版的安装程序 PyQt-gpl-4.2-Py2.5-Qt4.2.3.exe。

(8) 双击安装程序，按照默认设置直接安装即可。

(9) 完成安装后，需要将 Qt 的安装路径添加到 path 环境变量中。假设 Qt 安装在“D:\Qt\4.2.3”，则需将“D:\Qt\4.2.3\bin”目录添加到 path 环境变量中。重新启动系统使环境变量生效。

在设置 MinGW 的过程中，Qt 的安装程序可以发出警告，由于安装 Qt 仅使用其运行时库，所以可以忽略警告。另外，如果 Qt 安装程序没能自动安装 MinGW，可以到 MinGW 官方网站 <http://www.mingw.org> 下载 mingw-runtime-3.8.tar.gz 文件，将压缩包中 bin 目录下的

mingwm10.dll 文件复制到 “D:\Qt\4.2.3\bin” 目录中即可。

15.1.2 使用 PyQt 创建窗口

使用 PyQt 创建 GUI 脚本应首先创建一个 QtGui.QApplication 对象，并向其传递命令行参数。因此在脚本中除了导入 PyQt 模块外还应该导入 sys 模块。在脚本的最后应调用 QtGui.QApplication 对象的 exec_ 方法进入消息循环。如下所示的 HelloPyQt.py 创建了一个简单的窗口。

<pre># -*- coding:utf-8 -*- # file: HelloPyQt.py # import sys from PyQt4 import QtCore, QtGui class MyWindow(QtGui.QMainWindow): def __init__(self): QtGui.QMainWindow.__init__(self) self.setWindowTitle('PyQt') self.resize(300,200) app = QtGui.QApplication(sys.argv) mywindow = MyWindow() mywindow.show() app.exec_()</pre>	<pre># 导入 sys 模块 # 导入 PyQt 模块 # 通过继承 QtGui.QMainWindow 创建类 # 初始化方法 # 调用父类初始化方法 # 设置窗口标题 # 设置窗口大小 # 创建 QApplication 对象 # 创建 MyWindow 对象 # 显示窗口 # 进入消息循环</pre>
---	--

运行 HelloPyQt.py 脚本后将创建如图 15-5 所示的窗口。

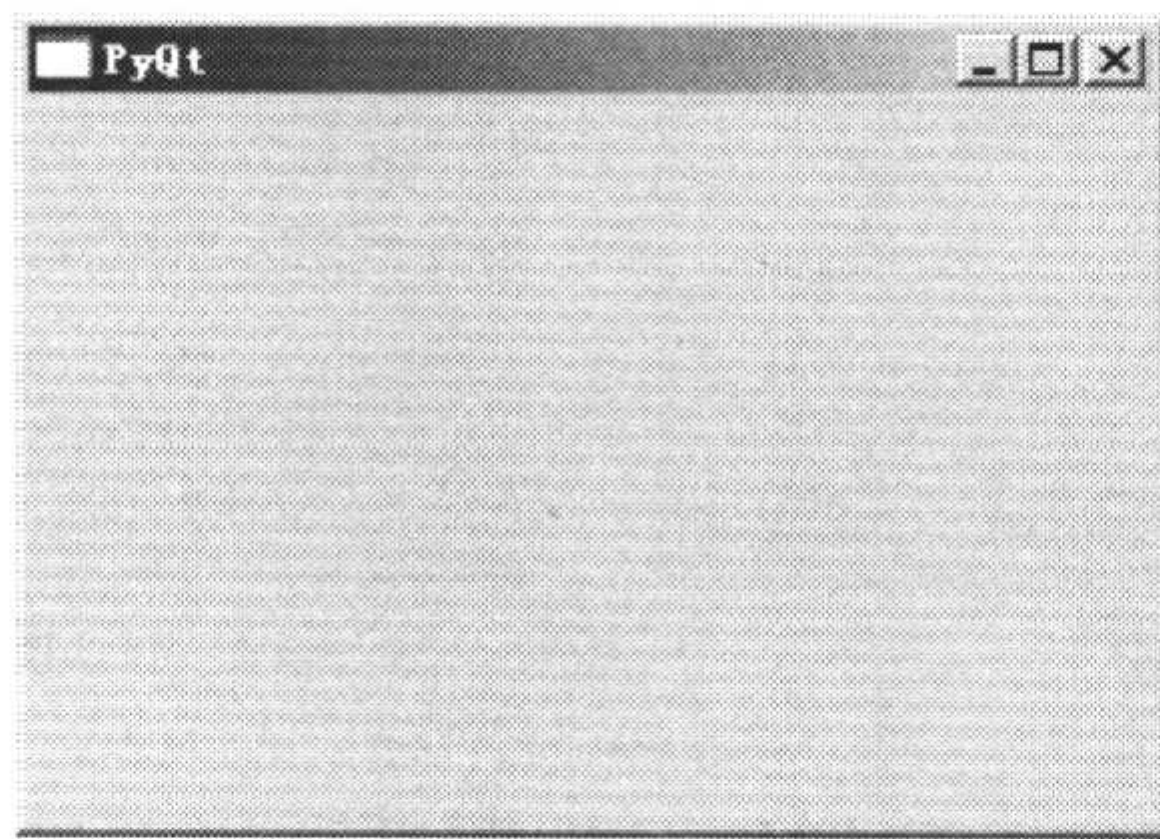


图 15-5 使用 PyQt 创建窗口

15.2 组件

PyQt 提供了丰富的组件进行 GUI 编程。在 PyQt 中可以方便地使用组件，并使用信号/插槽进行组件之间的通信，处理组件事件。

15.2.1 标签

在 PyQt 中使用 QtGui.QLabel 可以创建标签。使用其 setText 方法可以设置标签中的文字。使用 setTextFormat 方法可以设置标签中文字的格式。当创建标签后可以使用 QMainWindow

的 `setCentralWidget` 方法将标签添加到窗口中。其常用的方法有以下几个。

- `setPicture()`: 设置标签中的图片。
- `setText()`: 设置标签中的文字。
- `setTextFormat()`: 设置标签中文本的格式。
- `setAlignment()`: 设置标签中文本的对齐方式。

如下所示的 `PyQtLabel.py` 脚本创建了一个标签。

```
# -*- coding:utf-8 -*-
# file: PyQtLabel.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        label = QtGui.QLabel('PyQt\nLabel')
        label.setAlignment(QtCore.Qt.AlignCenter)
        self.setCentralWidget(label)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

导入 sys 模块
导入 PyQt 模块
通过继承 QtGui.QMainWindow 创建类
初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建标签
设置标签文字对齐样式
向窗口中添加标签
创建 QApplication 对象
创建 MyWindow 对象
显示窗口
进入消息循环

运行 `PyQtLabel.py` 脚本后将创建如图 15-6 所示的窗口。

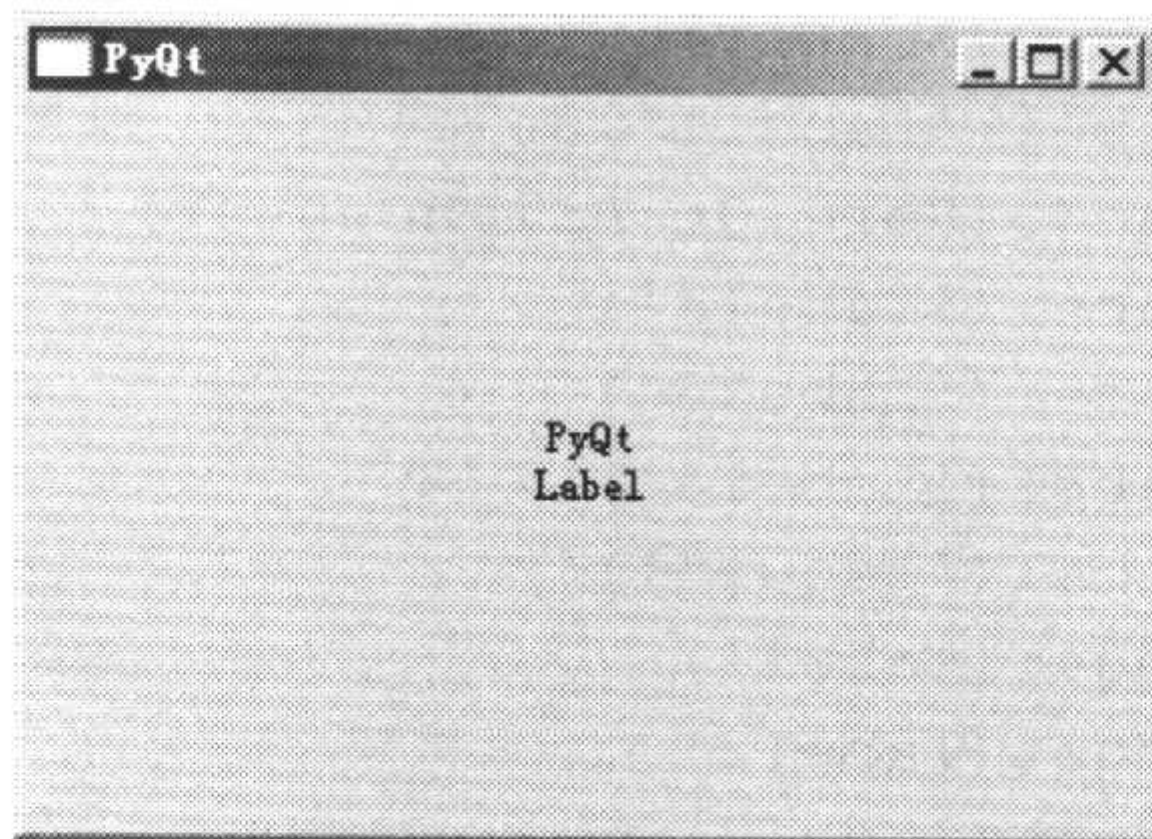


图 15-6 使用标签

15.2.2 布局组件和空白项

由于在窗口中使用 `setCentralWidget` 只能添加一个组件。如果需要向窗口中添加多个组件可以使用布局组件。使用布局组件不仅可以容纳多个组件，还可以设置组件的位置。空白项用于占位，配合布局组件可以更好地控制界面布局。

1. 布局组件

布局组件主要用于控制其内部组件的大小、位置等。布局组件可以包含其他的组件，也

可以包含其他的布局组件。常用的布局组件有以下几个。

- QLayout：基本的布局组件，只能被继承。
- QHBoxLayout：横向 Box 布局组件。
- QVBoxLayout：竖向 Box 布局组件。
- QGridLayout：Grid 布局组件。

布局组件共有的方法有以下几个。

- addWidget()：添加组件。
- addLayout()：添加其他布局组件。

如下所示的 PyQtLayout.py 脚本使用布局组件布置标签。

```
# -*- coding:utf-8 -*-
# file: PyQtLayout.py
#
import sys
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        hboxlayout1 = QtGui.QHBoxLayout()
        hboxlayout2 = QtGui.QHBoxLayout()
        vboxlayout1 = QtGui.QVBoxLayout()
        vboxlayout2 = QtGui.QVBoxLayout()
        label1 = QtGui.QLabel('Label1',self)
        label1.setAlignment(QtCore.Qt.AlignCenter)
        label2 = QtGui.QLabel('Label2')
        label3 = QtGui.QLabel('Label3')
        label4 = QtGui.QLabel('Label4')
        label5 = QtGui.QLabel('Label5')
        hboxlayout1.addWidget(label1)
        vboxlayout1.addWidget(label2)
        vboxlayout1.addWidget(label3)
        vboxlayout2.addWidget(label4)
        vboxlayout2.addWidget(label5)
        hboxlayout2.addLayout(vboxlayout1)
        hboxlayout2.addLayout(vboxlayout2)
        gridlayout.addLayout(hboxlayout1, 0 ,0)
        gridlayout.addLayout(hboxlayout2, 1, 0)
        gridlayout.setRowMinimumHeight (1, 180)
        self.setLayout(gridlayout)

app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
```

导入 sys 模块

导入 PyQt 模块

通过继承 QtGui.QWidget 创建类

初始化方法

调用父类初始化方法

设置窗口标题

设置窗口大小

创建布局组件

创建标签

向布局组件中添加标签

向 hboxlayout2 中添加 vboxlayout1

向 hboxlayout2 中添加 vboxlayout2

向第一行第一列中添加 hboxlayout1

向第二行第一列中添加 hboxlayout2

设置第二行的最小高度为 180

创建 QApplication 对象

创建 MyWindow 对象

显示窗口


```
app.exec_()
```

```
# 进入消息循环
```

运行 PyQtLayout.py 脚本后将创建如图 15-7 所示的窗口。

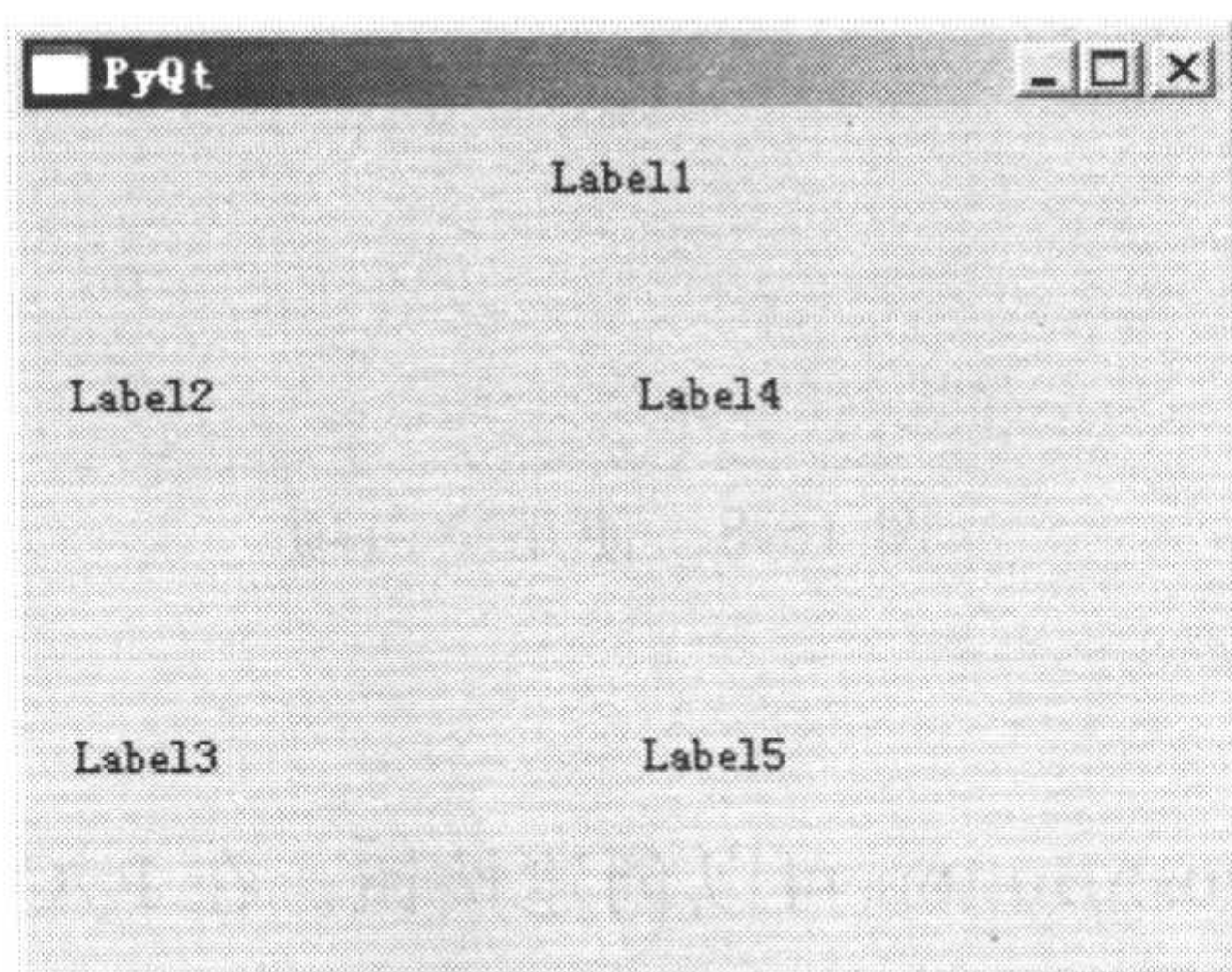


图 15-7 使用布局组件

2. 空白项

PyQt 中的空白项可以占据位置，这样就可以更好地布置其他的组件。使用 QtGui.QSpacerItem 可以创建空白项，可以使用宽度和高度设置空白项的大小。当创建空白项后需使用布局组件的 addItem 方法将空白项添加到布局组件中。如下所示的 PyQtSpacer.py 脚本使用空白项布置组件。

```
# -*- coding:utf-8 -*-
# file: PyQtSpacer.py
#
import sys
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        spacer1 = QtGui.QSpacerItem(300,40)
        spacer2 = QtGui.QSpacerItem(300,40)
        label = QtGui.QLabel('Label',self)
        label.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addItem(spacer1, 0 ,0)
        gridlayout.addWidget(label, 1, 0)
        gridlayout.addItem(spacer2, 2, 0)
        self.setLayout(gridlayout)

app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

```
# 导入 sys 模块
# 导入 PyQt 模块
# 通过继承 QtGui.QWidget 创建类
# 初始化方法
# 调用父类初始化方法
# 设置窗口标题
# 设置窗口大小
# 创建布局组件
# 创建空白项

# 创建标签
# 添加空白项
# 添加标签
# 添加空白项

# 创建 QApplication 对象
# 创建 MyWindow 对象
# 显示窗口
# 进入消息循环
```

运行 PyQtSpacer.py 脚本后将创建如图 15-8 所示的窗口。

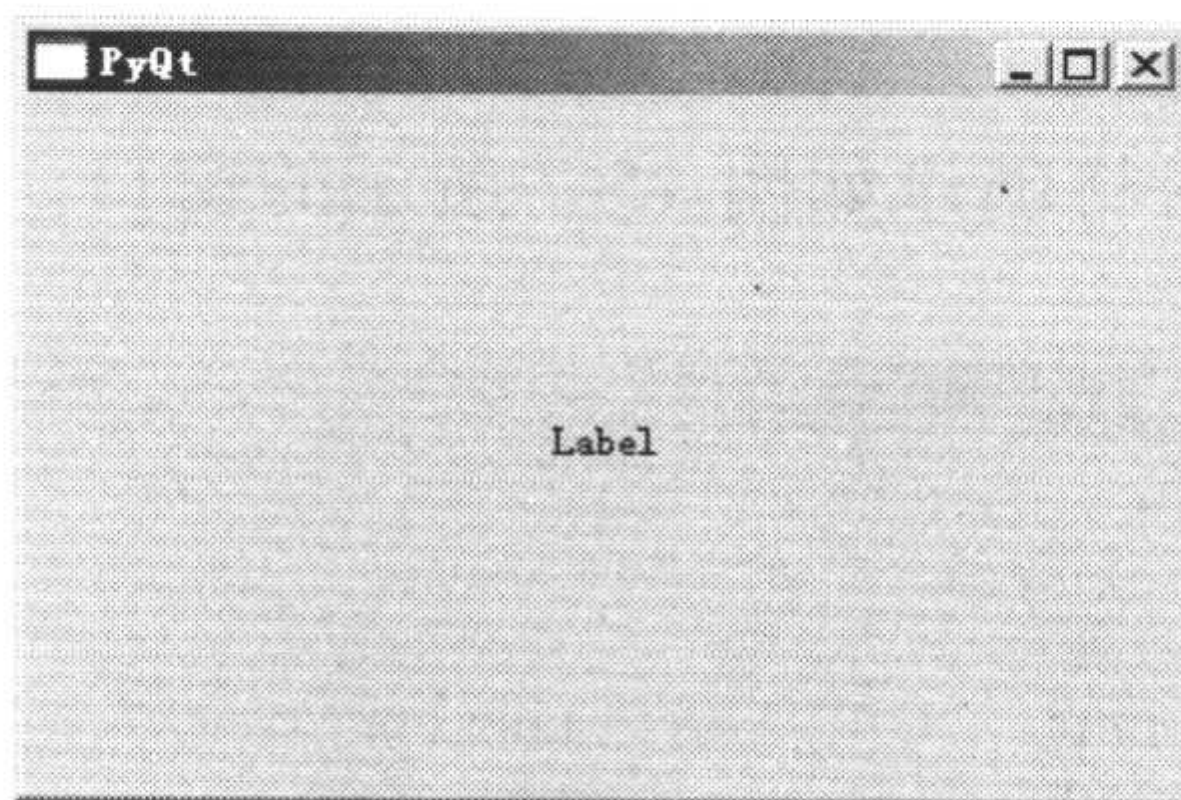


图 15-8 使用空白项

15.2.3 按钮

使用 PyQt 中的 QtGui.QPushButton 可以创建按钮。在 PyQt 中按钮事件是以信号/插槽的形式进行处理的，将按钮事件绑定到类的方法上。

1. 创建按钮

当使用 QtGui.QPushButton 创建按钮后可以使用以下几种方法设置按钮的样式、属性等。

- setDefault(): 将按钮设置为默认按钮。
- setFlat(): 将按钮设置为平坦模式。
- setMenu(): 设置按钮关联的菜单。
- menu(): 获得按钮所关联的菜单。

如下所示的 PyQtButton.py 脚本创建了两个按钮。

```
# -*- coding:utf-8 -*-
# file: PyQtButton.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        button1 = QtGui.QPushButton('Button1')
        gridlayout.addWidget(button1, 1, 1, 1, 3)
        button2 = QtGui.QPushButton('Button2')
        button2.setFlat(True)
        gridlayout.addWidget(button2, 2, 2)
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

导入 sys 模块
导入 PyQt 模块
通过继承 QtGui.QWidget 创建类
初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建布局组件
生成 Button1
添加 Button1
生成 Button2
添加 Button2
向窗口中添加布局组件
创建 QApplication 对象
创建 MyWindow 对象
显示窗口
进入消息循环

运行 PyQtButton.py 脚本后将创建如图 15-9 所示的窗口。

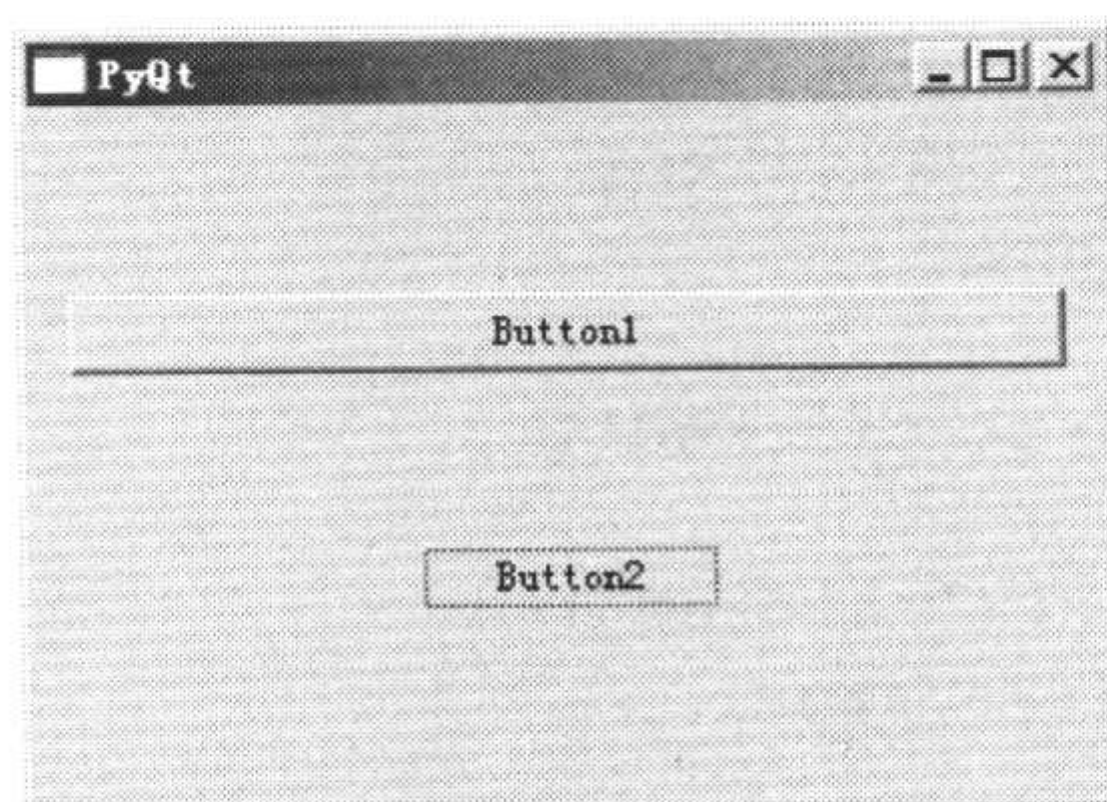


图 15-9 创建按钮

2. 信号和插槽

Qt 中的组件使用信号和插槽的形式来进行通信。Qt 的组件中有很多预定义的信号，当事件激发时，组件就发出信号。信号被发送给插槽进行处理。插槽实际上是处理特定信号的函数。在 PyQt 中也是一样，需要使用组件的 `connect` 方法将组件信号绑定到其处理插槽上。`connect` 方法的原型如下所示。

```
connect (QObject, SIGNAL(), SLOT(), Qt.ConnectionType)
```

其参数含义如下所示。

- `QObject`: 发送信号的组件。
- `SIGNAL()`: 组件所发送的信号。
- `SLOT()`: 插槽函数。
- `Qt.ConnectionType`: 可选参数，连接类型。

如下所示的 `PyQtButtonEvent.py` 脚本使用 `connect` 方法将按钮的 “`clicked()`” 信号连接到事件处理插槽函数上。

```
# -*- coding:utf-8 -*-
# file: PyQtButtonEvent.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        self.button1 = QtGui.QPushButton('Button1')
        gridlayout.addWidget(self.button1, 1, 1, 1, 3)
        self.button2 = QtGui.QPushButton('Button2')
        gridlayout.addWidget(self.button2, 2, 2)
        self.setLayout(gridlayout)
        self.connect(self.button1,
                     QtCore.SIGNAL('clicked()'),
                     self.OnButton1)
```

设置窗口标题

设置窗口大小

创建布局组件

生成 Button1

生成 Button2

向窗口中添加布局组件

Button1 事件

clicked() 信号

插槽函数


```

        self.connect(self.button2,
                      QtCore.SIGNAL('clicked()'),
                      self.OnButton2)
    def OnButton1(self):
        self.button1.setText('clicked')
    def OnButton2(self):
        self.button2.setText('clicked')
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()

```

```

# Button2 事件
# clicked() 信号
# 插槽函数
# Button1 插槽函数

# Button2 插槽函数

```

运行 PyQtButtonEvent.py 脚本后，单击窗口中的按钮，如图 15-10 所示。

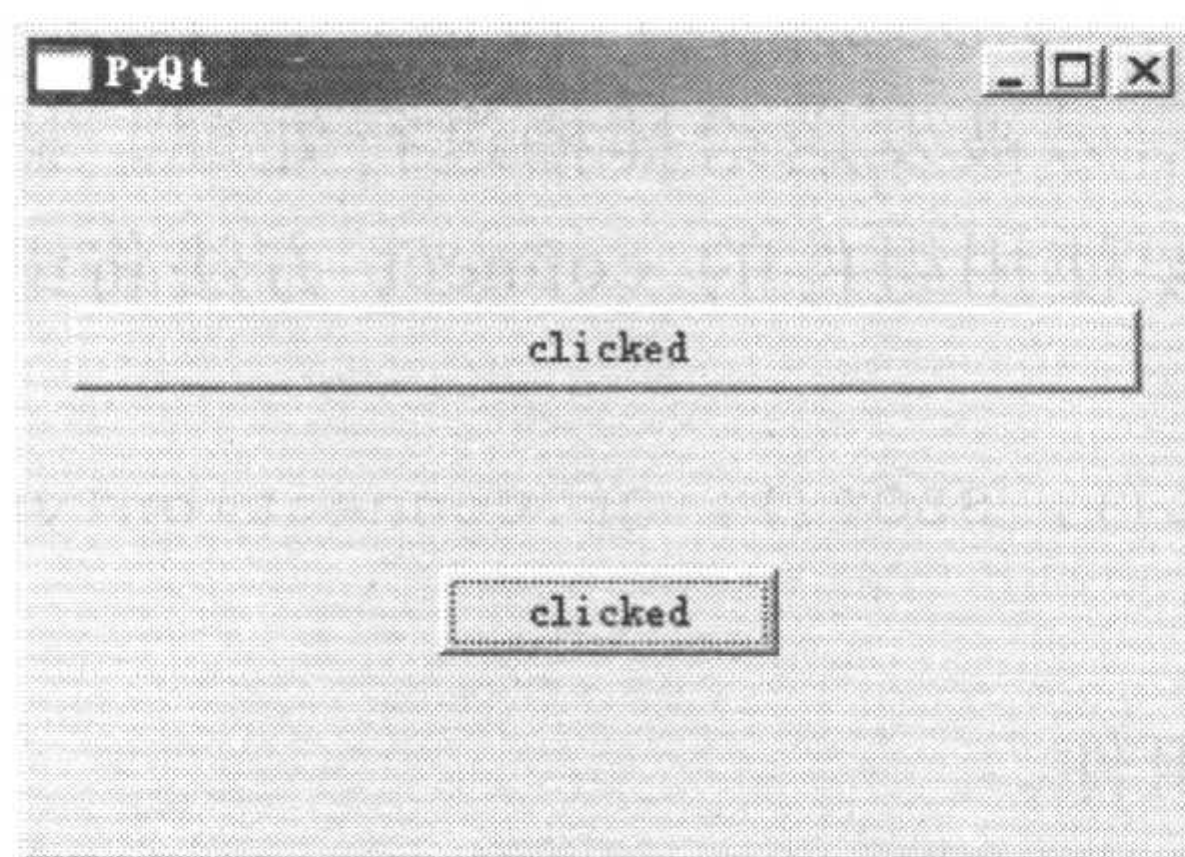


图 15-10 按钮事件

15.2.4 文本框

PyQt 中提供了单行文本框和多行文本框。使用 QtGui.QLineEdit 可以创建单行文本框，使用 QtGui.QTextEdit 可以创建多行文本框。

1. 单行文本框

使用 QtGui.QLineEdit 可以创建单行文本框，通过修改其属性可以将其设置为密码框。当创建单行文本框后，可以使用以下的方法设置文本框的属性或对文本框中的内容进行操作。

- clear(): 清除文本框中的内容。
- contextMenuEvent(): 右键菜单事件。
- copy(): 复制文本框中的内容。
- cut(): 剪切文本框中的内容。
- paste(): 向文本框中粘贴内容。
- redo(): 重做。
- selectAll(): 全选。
- selectedText(): 获得选中的文本。

- `setAlignment()`: 设置文本对齐方式。
- `setEchoMode()`: 设置文本框类型。
- `setMaxLength()`: 设置文本框中最大字符数。
- `setText()`: 设置文本框中的文字。
- `undo()`: 撤销。

如下所示的 `PyQtLineEdit.py` 脚本创建了一个单行文本框和密码框。

```
# -*- coding:utf-8 -*-
# file: PyQtLineEdit.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        labell = QtGui.QLabel('Normal Lineedit')
        labell.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addWidget(labell, 0, 0 )
        edit1 = QtGui.QLineEdit()
        gridlayout.addWidget(edit1, 1, 0)
        label2 = QtGui.QLabel('Password')
        label2.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addWidget(label2, 2, 0)
        edit2 = QtGui.QLineEdit()
        edit2.setEchoMode(QtGui.QLineEdit.Password)
        gridlayout.addWidget(edit2, 3, 0)
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

通过继承 QtGui.QWidget 创建类
初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建布局组件
创建标签

创建单行文本框

创建标签

创建单行文本框
将其设置为密码框

创建 QApplication 对象
创建 MyWindow 对象
显示窗口
进入消息循环

运行 `PyQtLineEdit.py` 脚本后在单行文本框中输入字符，如图 15-11 所示。

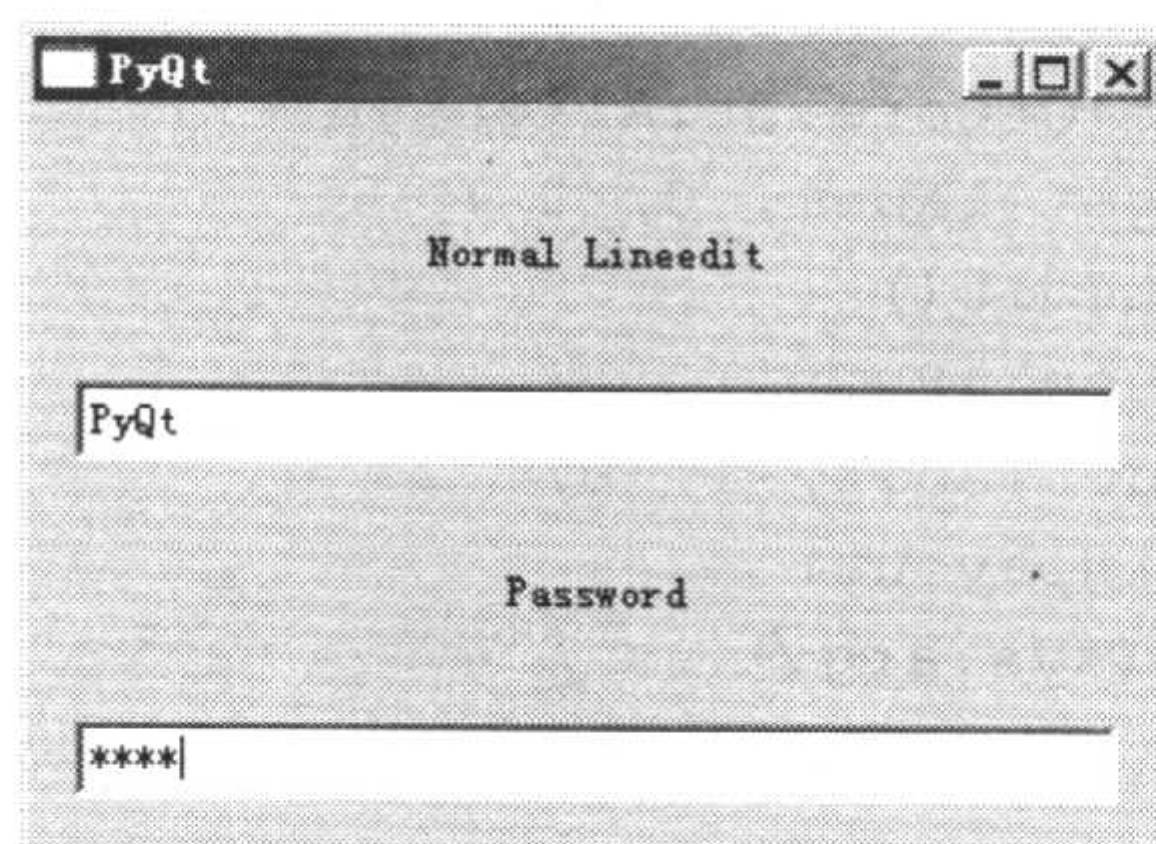


图 15-11 创建单行文本框

2. 多行文本框

使用 QtGui.QTextEdit 可以创建多行文本框。当创建多行文本框后，可以使用以下的方法设置文本框的属性或对文本框中的内容进行操作。

- append(): 向文本框中追加内容。
- clear(): 清除文本框中的内容。
- contextMenuEvent(): 右键菜单事件。
- copy(): 复制文本框中的内容。
- cut(): 剪切文本框中的内容。
- find(): 查找文本。
- paste(): 向文本框中粘贴内容。
- redo(): 重做。
- selectAll(): 全选。
- selectedText(): 获得选中的文本。
- setAlignment(): 设置文本对齐方式。
- setText(): 设置文本框中的文字。
- undo(): 撤销。

如下所示的 PyQtTextEdit.py 脚本创建了一个多行文本框。

```
# -*- coding:utf-8 -*-
# file: PyQtTextEdit.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        label = QtGui.QLabel('TextEdit')
        label.setAlignment(QtCore.Qt.AlignCenter)
        gridlayout.addWidget(label, 0, 0 )
        edit = QtGui.QTextEdit()
        edit.setText('Python\nPyQt')
        gridlayout.addWidget(edit, 1, 0)
        self.setLayout(gridlayout)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

```
# 初始化方法
# 调用父类初始化方法
# 设置窗口标题
# 设置窗口大小
# 创建布局组件
# 创建标签

# 创建多行文本框
# 设置文本框中的文字

# 创建 QApplication 对象
# 创建 MyWindow 对象
# 显示窗口
# 进入消息循环
```

运行 PyQtTextEdit.py 脚本后将创建如图 15-12 所示的窗口。

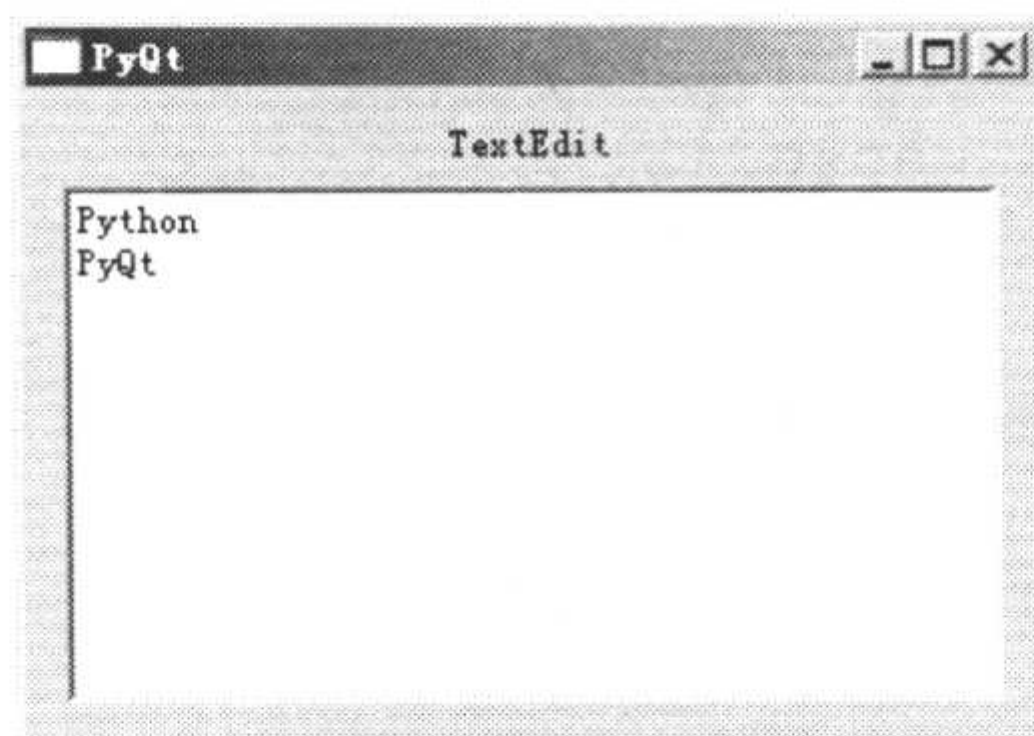


图 15-12 创建多行文本框

15.2.5 单选框和复选框

在 PyQt 中使用 QtGui.QRadioButton 可以创建单选框，使用 QtGui.QCheckBox 可以创建复选框。单选框和复选框都可以通过 setChecked() 设置状态，通过 isChecked() 方法获取状态。如下所示的 PyQtRCButton.py 脚本创建了一组单选框和一个复选框。

```
# -*- coding:utf-8 -*-
# file: PyQtRCButton.py
#
import sys
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        self.label1 = QtGui.QLabel('Label1')
        self.label2 = QtGui.QLabel('Label2')
        gridlayout.addWidget(self.label1, 1, 2)
        gridlayout.addWidget(self.label2, 2, 2)
        self.radio1 = QtGui.QRadioButton('Radio1')
        self.radio2 = QtGui.QRadioButton('Radio2')
        self.radio3 = QtGui.QRadioButton('Radio3')
        self.radio1.setChecked(True)
        gridlayout.addWidget(self.radio1, 1, 1)
        gridlayout.addWidget(self.radio2, 2, 1)
        gridlayout.addWidget(self.radio3, 3, 1)
        self.check = QtGui.QCheckBox('check')
        self.check.setChecked(True)
        gridlayout.addWidget(self.check, 3, 2)
        self.button = QtGui.QPushButton('Test')
        gridlayout.addWidget(self.button, 4, 1, 1, 2)
        self.setLayout(gridlayout)
        self.connect(self.button,
                     QtCore.SIGNAL('clicked()'),
                     self.OnButton)
```

初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建布局组件
创建标签

创建单选框

将 Radio1 选中
添加单选框

创建复选框
将复选框选中

创建按钮

向窗口中添加布局组件
按钮事件


```
def OnButton(self):
    if self.radio1.isChecked():
        self.label1.setText('Radio1')
    elif self.radio2.isChecked():
        self.label1.setText('Radio2')
    else :
        self.label1.setText('Radio3')
    if self.check.isChecked():
        self.label2.setText('checked')
    else:
        self.label2.setText('uncheck')
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

按钮插槽函数
判断单选框是否被选中
判断复选框是否被选中

运行 PyQtRCButton.py 脚本后，单击【Test】按钮，将根据单选框和复选框的状态设置标签中的文本，如图 15-13 所示。

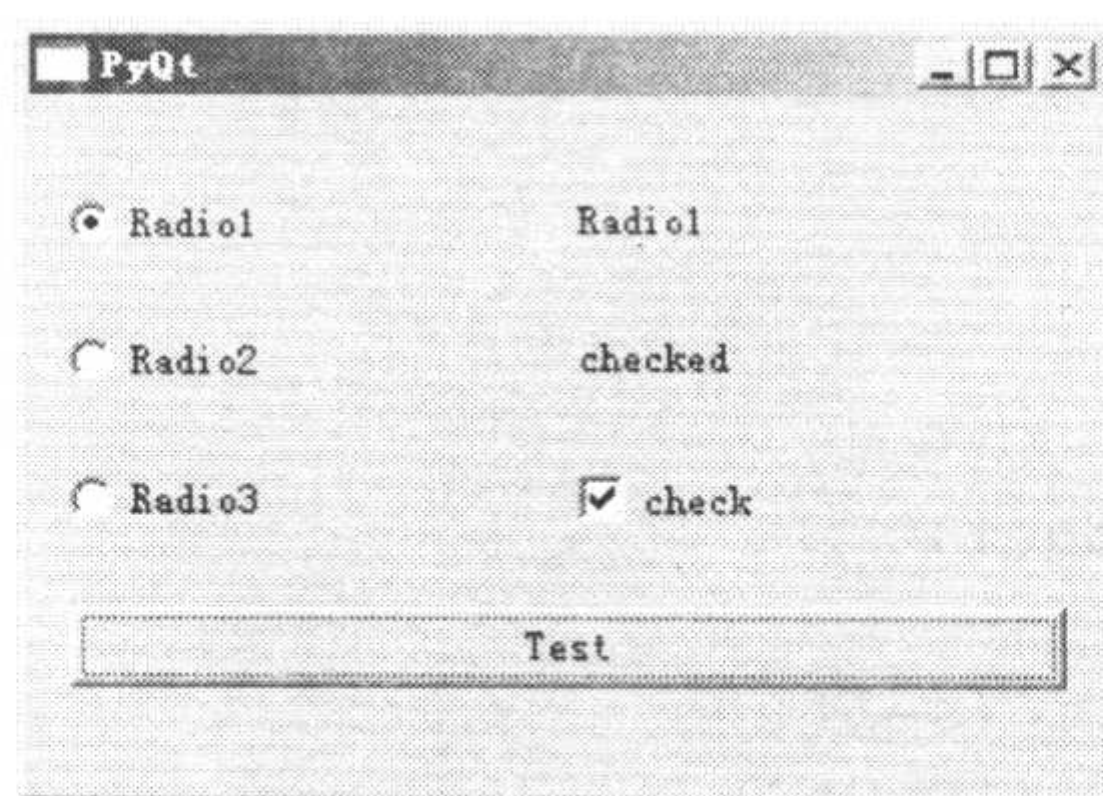


图 15-13 单选框和复选框

15.2.6 菜单

在 PyQt 中可以使用 QMenuBar 创建菜单条，使用 QMenu 创建菜单。菜单单击事件也可以像按钮事件一样通过信号/插槽的形式绑定到类的方法上。

1. 创建菜单

当使用 QMenuBar 创建菜单条后，可以使用其 addMenu 方法向其中添加菜单。然后通过菜单的 addAction 方法向菜单中添加菜单命令。对于 QtGui.QMainWindow 可以直接使用其 menuBar 方法获得其菜单条，而无需使用 QMenuBar 创建菜单条。如下所示的 PyQtMenu.py 脚本创建了一组菜单。

```
# -*- coding:utf-8 -*-
# file: PyQtMenu.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
```

通过继承 QtGui.QMainWindow 创建类
初始化方法
调用父类初始化方法


```

self.setWindowTitle('PyQt')
self.resize(300,200)
menubar = self.menuBar()
file = menubar.addMenu('&File')
file.addAction('Open')
file.addAction('Save')
file.addSeparator()
file.addAction('Close')
edit = menubar.addMenu('&Edit')
edit.addAction('Copy')
edit.addAction('Paste')
edit.addAction('Cut')
edit.addAction('SelectAll')
help = menubar.addMenu('&Help')
help.addAction('About')

app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()

```

设置窗口标题
 # 设置窗口大小
 # 获得窗口的菜单条
 # 添加 File 菜单
 # 添加 Open 命令
 # 添加 Save 命令
 # 添加分隔符
 # 添加 Close 命令
 # 添加 Edit 菜单
 # 添加 Copy 命令
 # 添加 Paste 命令
 # 添加 Cut 命令
 # 添加 SelectAll 命令
 # 添加 Help 菜单
 # 添加 About 命令
 # 创建 QApplication 对象
 # 创建 MyWindow 对象
 # 显示窗口
 # 进入消息循环

运行 PyQtMenu.py 脚本后将创建如图 15-14~图 15-16 所示的菜单。

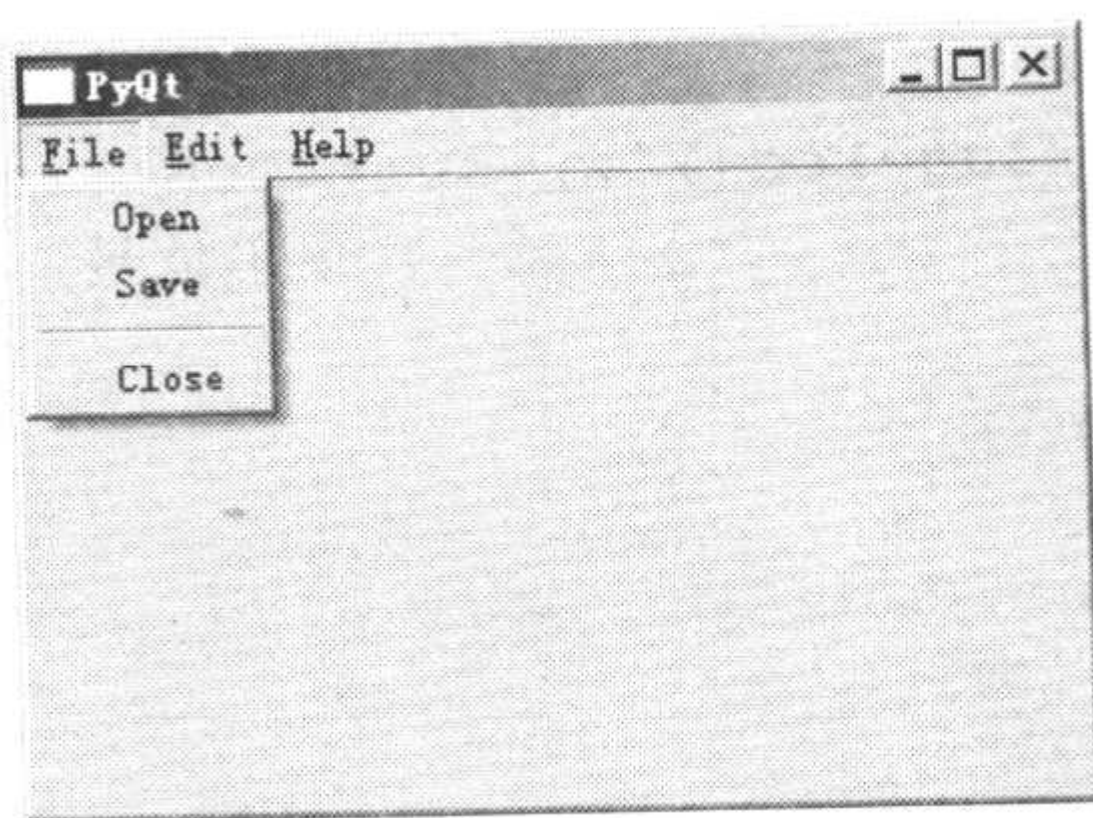


图 15-14 File 菜单

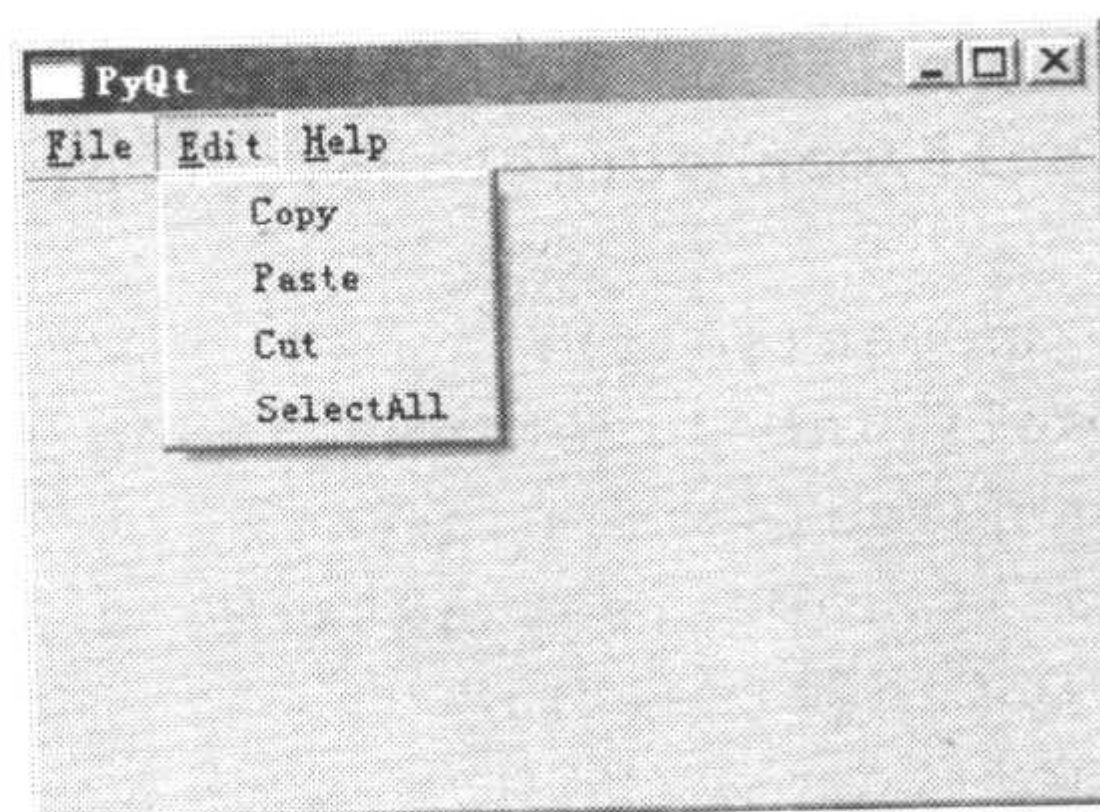


图 15-15 Edit 菜单

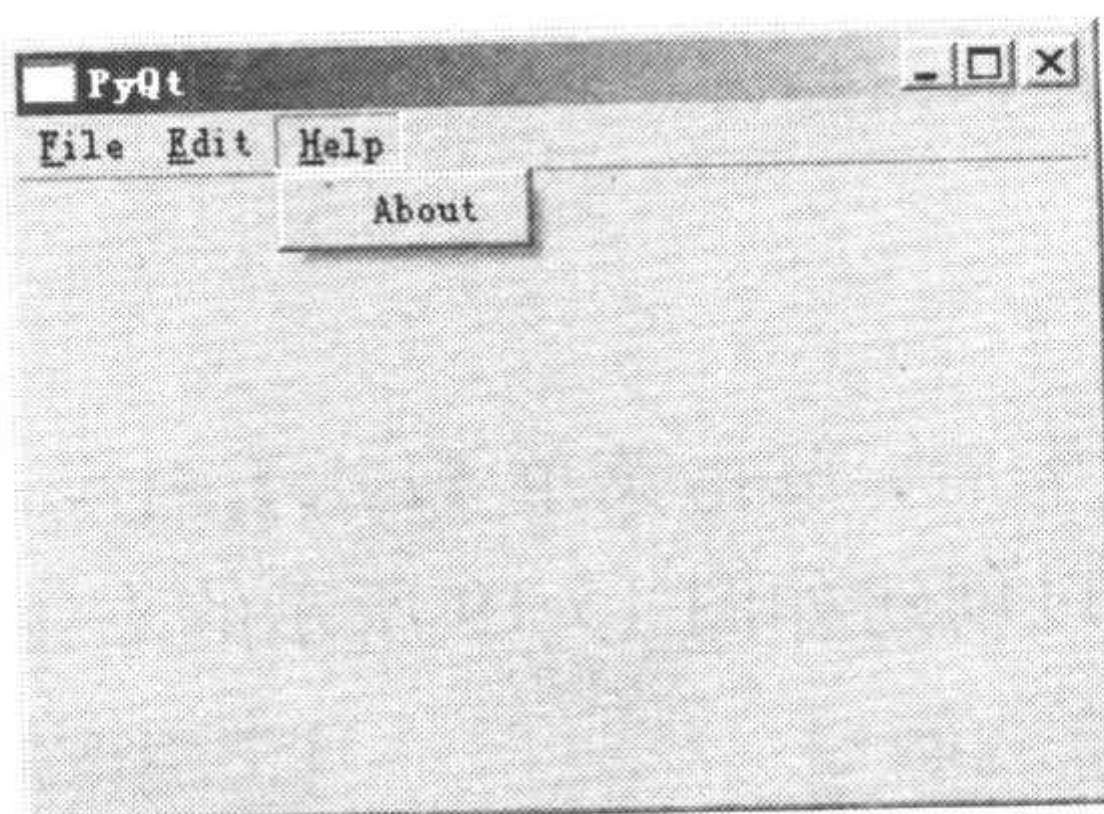


图 15-16 Help 菜单

2. 菜单事件

菜单事件的处理和按钮事件的处理一样，使用窗口的 connect 方法将菜单信号绑定到插槽事件处理函数上。不同的是对于菜单事件，通常需要绑定其“triggered()”信号。如下所示的 PyQtMenuAction.py 对菜单事件进行处理，并且创建了右键菜单。

```
# -*- coding:utf-8 -*-
```



```
# self.file: PyQtMenuAction.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        self.label = QtGui.QLabel('Menu Action')
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.setCentralWidget(self.label)
        menubar = self.menuBar()
        self.file = menubar.addMenu('&File')
        open = self.file.addAction('Open')
        self.connect(open, QtCore.SIGNAL('triggered()'), self.OnOpen)

        save = self.file.addAction('Save')
        self.connect(save, QtCore.SIGNAL('triggered()'), self.OnSave)

        self.file.addSeparator()
        close = self.file.addAction('Close')
        self.connect(close, QtCore.SIGNAL('triggered()'), self.OnClose)

    def OnOpen(self):
        self.label.setText('Menu Action: Open')
    def OnSave(self):
        self.label.setText('Menu Action: Save')
    def OnClose(self):
        self.close()
    def contextMenuEvent(self, event):
        self.file.exec_(event.globalPos())
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建标签
设置标签文字对齐样式

获得窗口的菜单条
添加 File 菜单
添加 Open 命令
菜单信号
添加 Save 命令
菜单信号
添加分隔符
添加 Close 命令
菜单信号

重载弹出式菜单事件
创建 QApplication 对象
创建 MyWindow 对象
显示窗口
进入消息循环

运行 PyQtMenuAction.py 脚本后，单击菜单【File】命令，如图 15-17 所示。单击【Open】命令，如图 15-18 所示。右击窗口创建如图 15-19 所示的右键菜单。

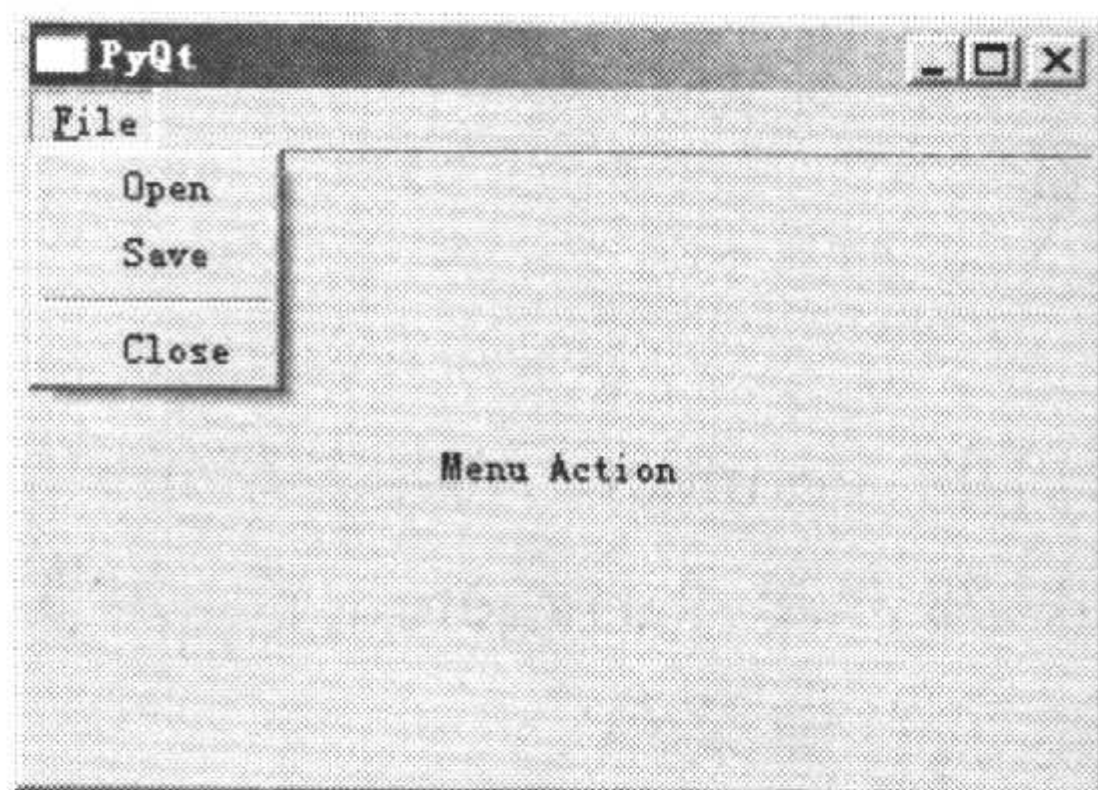


图 15-17 File 菜单

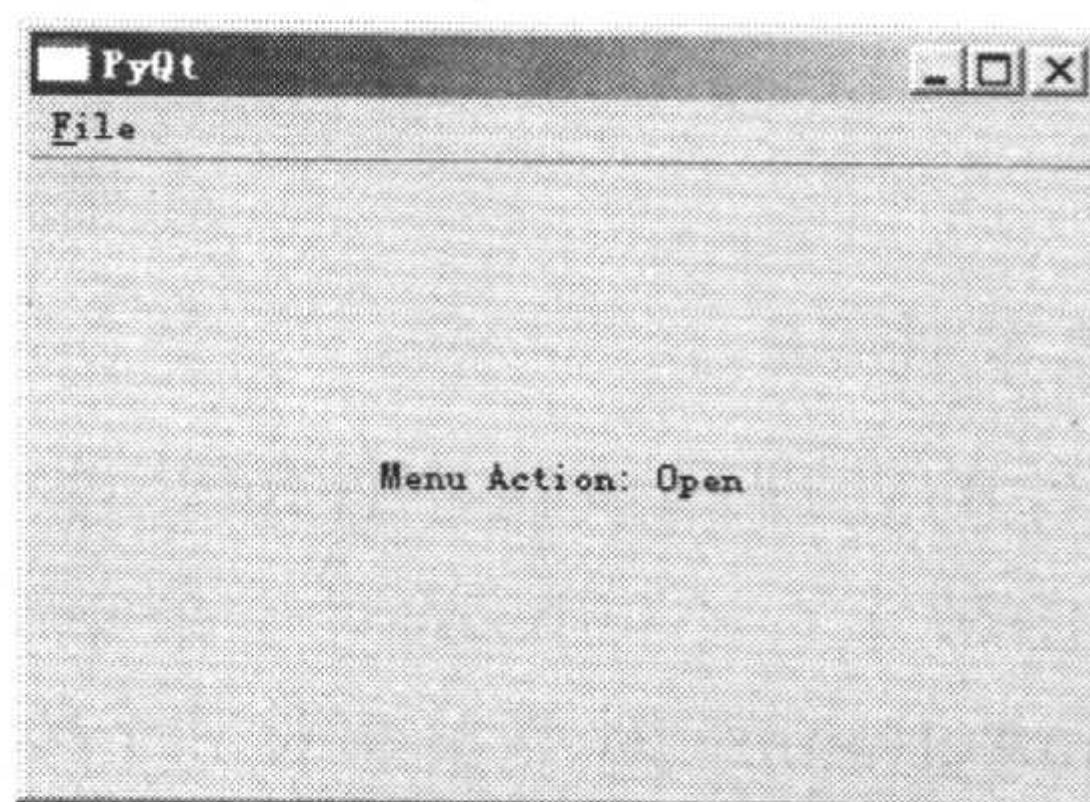


图 15-18 单击 Open 命令

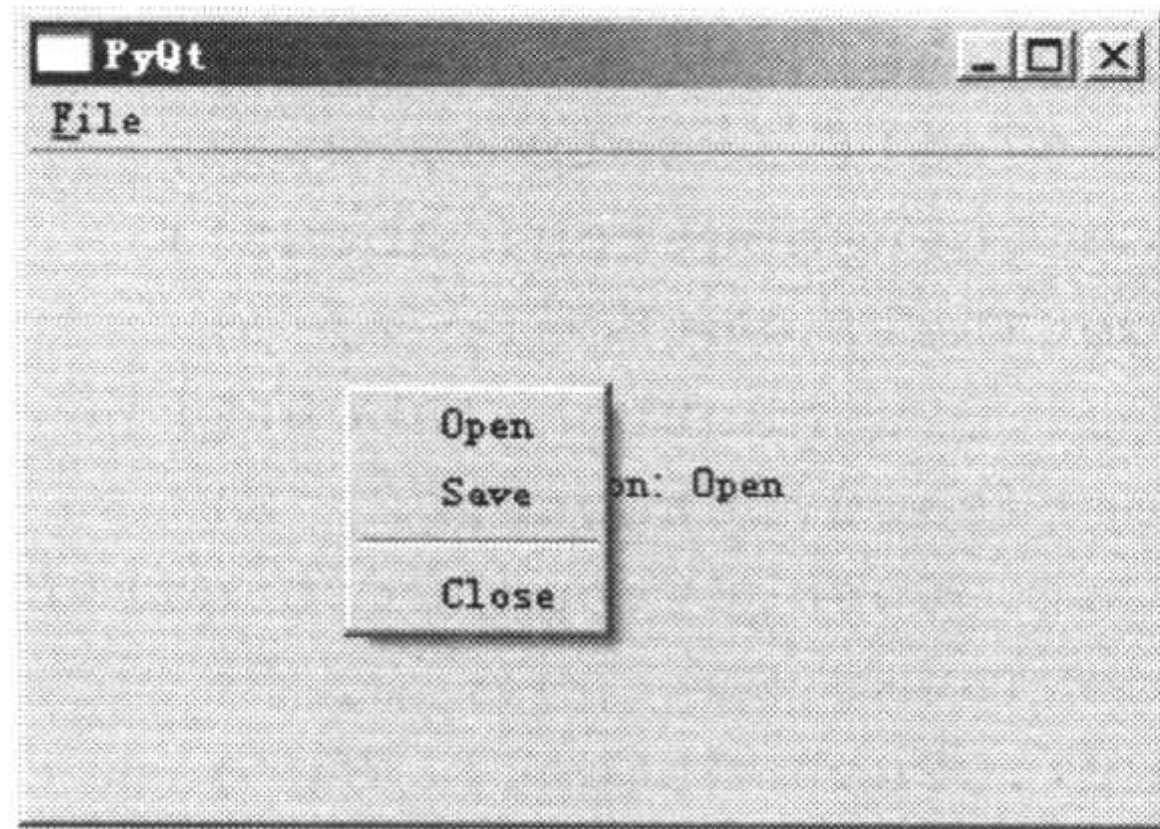


图 15-19 创建右键菜单

15.3 对话框

PyQt 中提供了基本的消息框和标准对话框。在 PyQt 中也可以根据需求创建自定义的对话框，并在对话框中使用 PyQt 中的组件。

15.3.1 消息框和标准对话框

使用 PyQt 提供的类和方法可以方便地创建和使用消息框、标准对话框等。标准对话框包括基本的打开、关闭文件对话框，字体选择对话框和颜色选择对话框等。

1. 消息框

使用 QtGui.QMessageBox 类中的方法可以创建简单的消息框，用于向用户传递信息。QtGui.QMessageBox 类中包含了以下几种创建消息框的方法。

- about(): 创建关于消息框。
- aboutQt(): 创建关于 Qt 消息框。
- critical(): 创建错误处理对话框。
- information(): 创建信息消息框。
- question(): 创建询问消息框。
- warning(): 创建警告消息框。

如下所示的 PyQtMessageBox.py 使用 QtGui.QMessageBox 创建消息框。

```
# -*- coding:utf-8 -*-
# file: PyQtMessageBox.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        # 初始化方法
        # 调用父类初始化方法
        # 设置窗口标题
        # 设置窗口大小
        # 创建布局组件
```



```

self.label = QtGui.QLabel('MessBox example')
gridlayout.addWidget(self.label, 1, 3, 1, 3)
self.button1 = QtGui.QPushButton('About')
gridlayout.addWidget(self.button1, 2, 1)
self.button2 = QtGui.QPushButton('AboutQt')
gridlayout.addWidget(self.button2, 2, 2)
self.button3 = QtGui.QPushButton('Critical')
gridlayout.addWidget(self.button3, 2, 3)
self.button4 = QtGui.QPushButton('Info')
gridlayout.addWidget(self.button4, 2, 4)
self.button5 = QtGui.QPushButton('Qusetion')
gridlayout.addWidget(self.button5, 2, 5)
self.button6 = QtGui.QPushButton('Warning')
gridlayout.addWidget(self.button6, 2, 6)
spacer = QtGui.QSpacerItem(200, 80)
gridlayout.addItem(spacer, 3, 1, 1, 5)
self.setLayout(gridlayout)
self.connect(self.button1,
             QtCore.SIGNAL('clicked()'),
             self.OnButton1)
self.connect(self.button2,
             QtCore.SIGNAL('clicked()'),
             self.OnButton2)
self.connect(self.button3,
             QtCore.SIGNAL('clicked()'),
             self.OnButton3)
self.connect(self.button4,
             QtCore.SIGNAL('clicked()'),
             self.OnButton4)
self.connect(self.button5,
             QtCore.SIGNAL('clicked()'),
             self.OnButton5)
self.connect(self.button6,
             QtCore.SIGNAL('clicked()'),
             self.OnButton6)
def OnButton1(self):
    self.button1.setText('clicked')
    QtGui.QMessageBox.about(self, 'PyQt', 'About')
def OnButton2(self):
    self.button2.setText('clicked')
    QtGui.QMessageBox.aboutQt(self, 'PyQt')
def OnButton3(self):
    self.button3.setText('clicked')
    r = QtGui.QMessageBox.critical(self, 'PyQt',
                                   'Critical',
                                   QtGui.QMessageBox.Abort,
                                   QtGui.QMessageBox.Retry,
                                   QtGui.QMessageBox.Ignore)
    if r == QtGui.QMessageBox.Abort:

```

生成 Button1

生成 Button2

生成 Button3

生成 Button4

生成 Button5

生成 Button6

向窗口中添加布局组件

Button1 事件

Button2 事件

Button3 事件

Button4 事件

Button5 事件

Button6 事件

Button1 插槽函数

创建 About 消息框

Button2 插槽函数

创建 AboutQt 消息框

Button3 插槽函数

创建 Critical 消息框

```

        self.label.setText('Abort')
    elif r == QtGui.QMessageBox.Retry:
        self.label.setText('Retry')
    else:
        self.label.setText('Ignore')
def OnButton4(self):                                # Button4 插槽函数
    self.button4.setText('clicked')
    QtGui.QMessageBox.information(self, 'PyQt', 'Information')
                                                    # 创建 Information 消息框
def OnButton5(self):                                # Button5 插槽函数
    self.button5.setText('clicked')
    r = QtGui.QMessageBox.question(self, 'PyQt',      # 创建 Question 消息框
        'Question',
        QtGui.QMessageBox.Yes,
        QtGui.QMessageBox.No,
        QtGui.QMessageBox.Cancel)
def OnButton6(self):                                # Button6 插槽函数
    self.button6.setText('clicked')
    r = QtGui.QMessageBox.warning(self, 'PyQt',      # 创建 Warning 消息框
        'Warning',
        QtGui.QMessageBox.Yes,
        QtGui.QMessageBox.No)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()

```

运行 PyQtMessageBox.py 脚本后, 分别右击窗口中的按钮, 创建如图 15-20~图 15-25 所示的窗口。

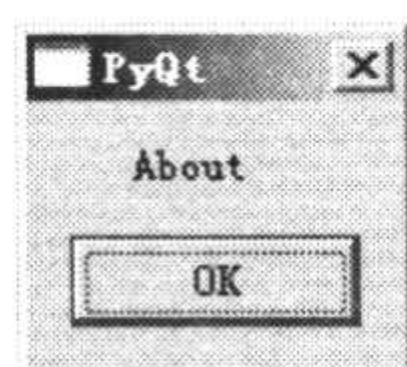


图 15-20 About 消息框

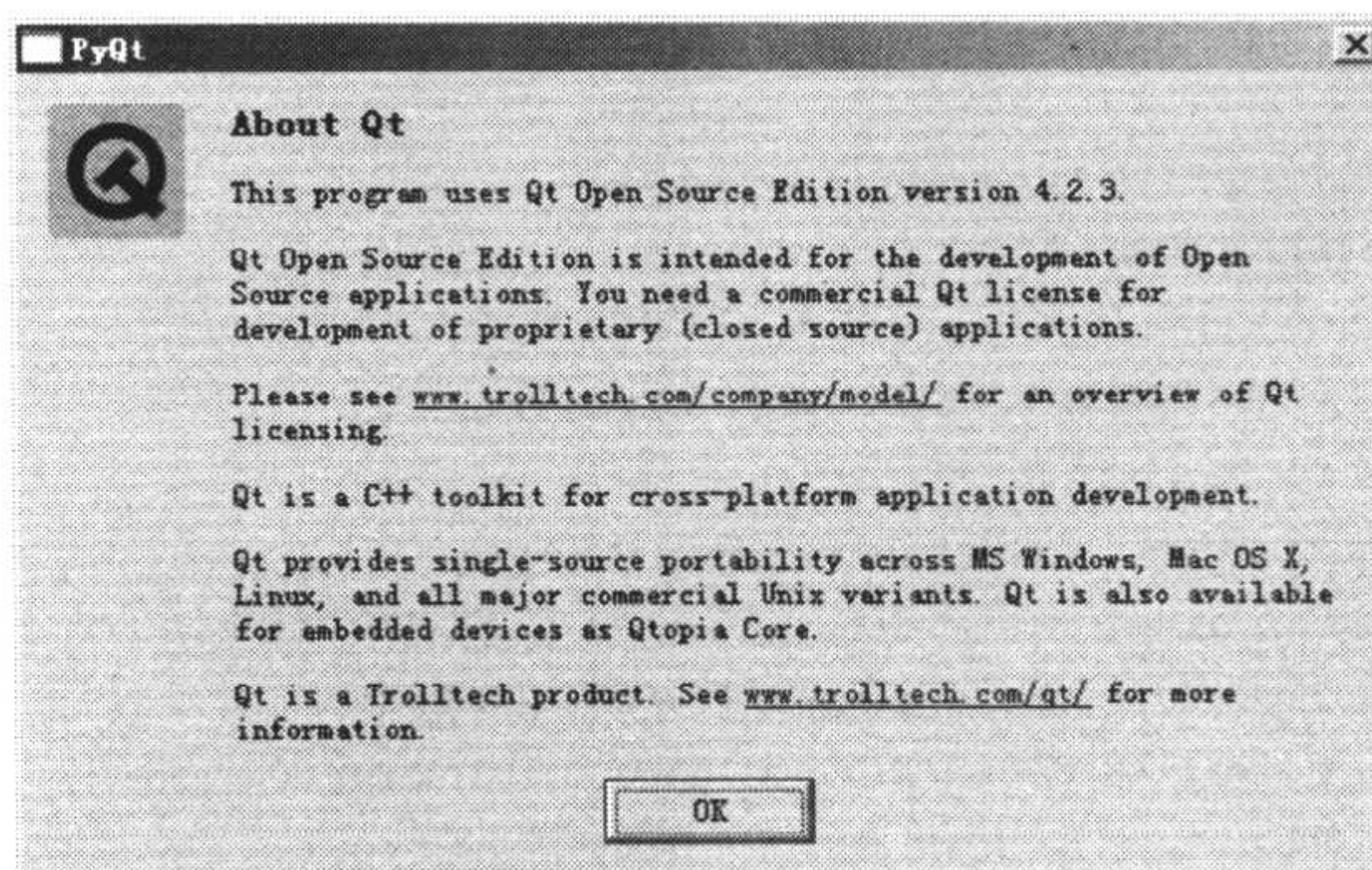


图 15-21 AboutQt 消息框

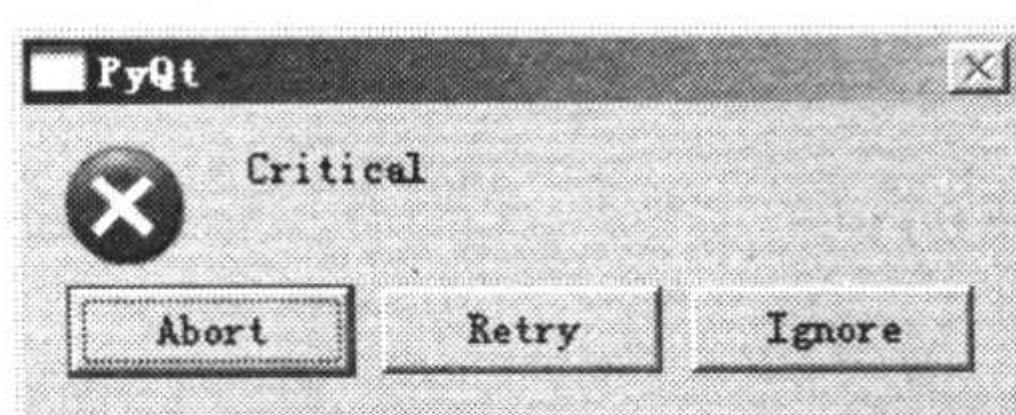


图 15-22 Cirtical 消息框

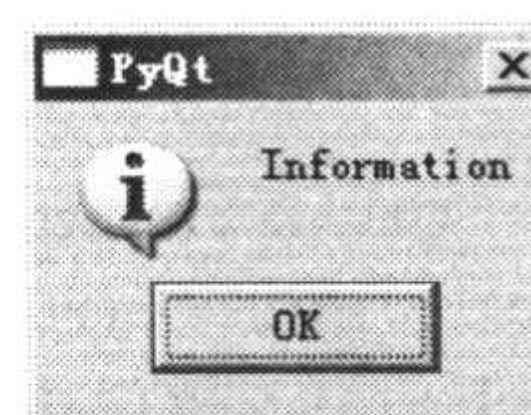


图 15-23 Information 消息框

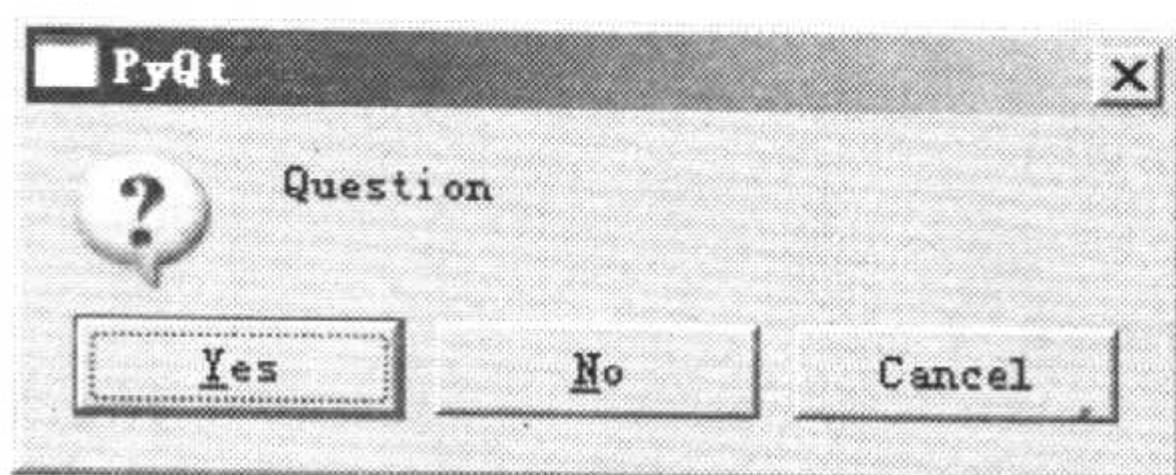


图 15-24 Question 消息框

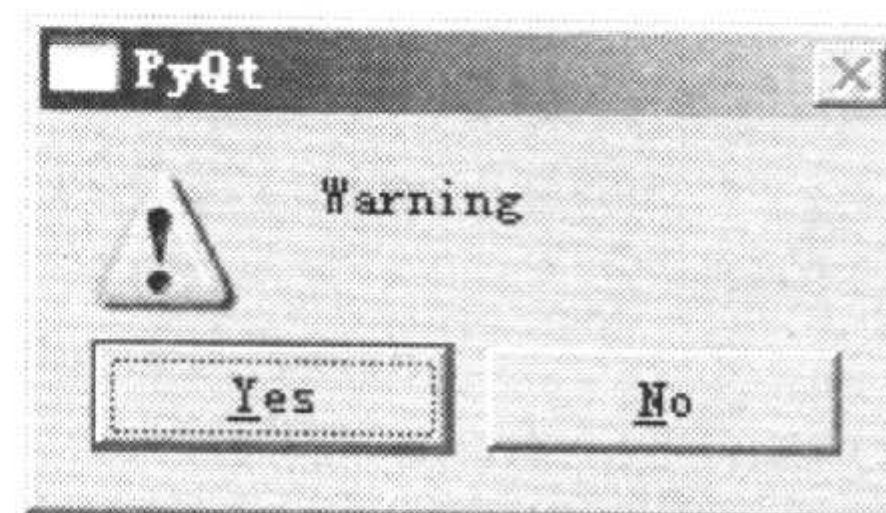


图 15-25 Warning 消息框

2. 标准对话框

在 PyQt 中使用 QtGui.QFileDialog 提供的方法可以创建文件打开、关闭对话框。使用 QtGui.QFontDialog 提供的方法可以创建字体选择对话框。使用 QtGui.QColorDialog 提供的方法可以创建颜色选择对话框。对于 QtGui.QFileDialog，其方法有如下几个。

- getExistingDirectory(): 创建选取路径对话框。
- getOpenFileName(): 创建打开文件对话框。
- getOpenFileNames(): 创建打开文件对话框，可以同时打开多个文件。
- getSaveFileName(): 创建保存文件对话框。

对于 QtGui.QfontDialog，其静态方法仅有 getFont，用于创建字体选择对话框。对于 QtGui.QcolorDialog，可以使用其 getColor 方法创建颜色选择对话框。如下所示的 PyQtStandarDialog.py 创建了 PyQt 提供的标准对话框。

```
# -*- coding:utf-8 -*-
# file: PyQtStandarDialog.py
#
import sys
from PyQt4 import QtCore, QtGui
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        self.label = QtGui.QLabel('StandarDialog example')
        gridlayout.addWidget(self.label, 1, 2)
        self.button1 = QtGui.QPushButton('File')
        gridlayout.addWidget(self.button1, 2, 1)
        self.button2 = QtGui.QPushButton('Font')
        gridlayout.addWidget(self.button2, 2, 2)
        self.button3 = QtGui.QPushButton('Color')
        gridlayout.addWidget(self.button3, 2, 3)
        spacer = QtGui.QSpacerItem(200, 80)
        gridlayout.addItem(spacer, 3, 1, 1, 3)
        self.setLayout(gridlayout)
        self.connect(self.button1,
```

初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建布局组件

生成 Button1

生成 Button2

生成 Button3

向窗口中添加布局组件
Button1 事件

第15章 使用PyQT编写GUI

```

        QtCore.SIGNAL('clicked()'),
        self.OnButton1)
self.connect(self.button2,
             QtCore.SIGNAL('clicked()'),
             self.OnButton2)
self.connect(self.button3,
             QtCore.SIGNAL('clicked()'),
             self.OnButton3)
def OnButton1(self):
    self.button1.setText('clicked')
    filename = QtGui.QFileDialog.getOpenFileName(self, 'Open')
    if not filename.isEmpty():
        self.label.setText(filename)
def OnButton2(self):
    self.button2.setText('clicked')
    font, ok = QtGui.QFontDialog.getFont()
    if ok:
        self.label.setText(font.key())
def OnButton3(self):
    self.button3.setText('clicked')
    color = QtGui.QColorDialog.getColor()
    if color.isValid():
        self.label.setText(color.name())
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()

```

Button2 事件

Button3 事件

创建文件打开对话框

Button2 插槽函数

创建字体选择对话框

Button3 插槽函数

创建颜色选择对话框

运行 PyQtStandarDialog.py 脚本后, 单击【File】按钮, 将创建如图 15-26 所示的打开文件对话框。单击【Font】按钮, 将创建如图 15-27 所示的字体选择对话框。单击【Color】按钮, 将创建如图 15-28 所示的颜色选择对话框。

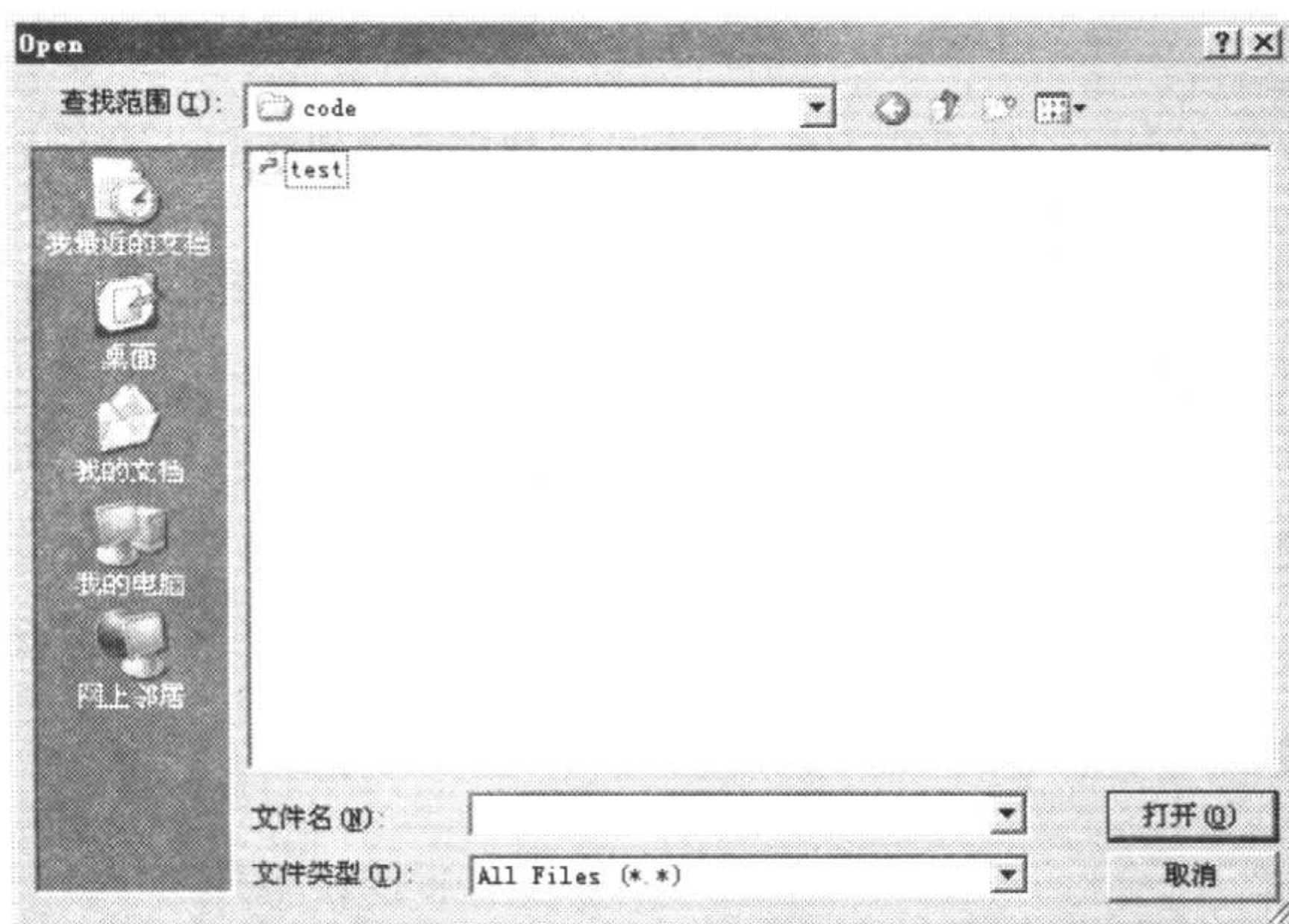


图 15-26 打开文件对话框

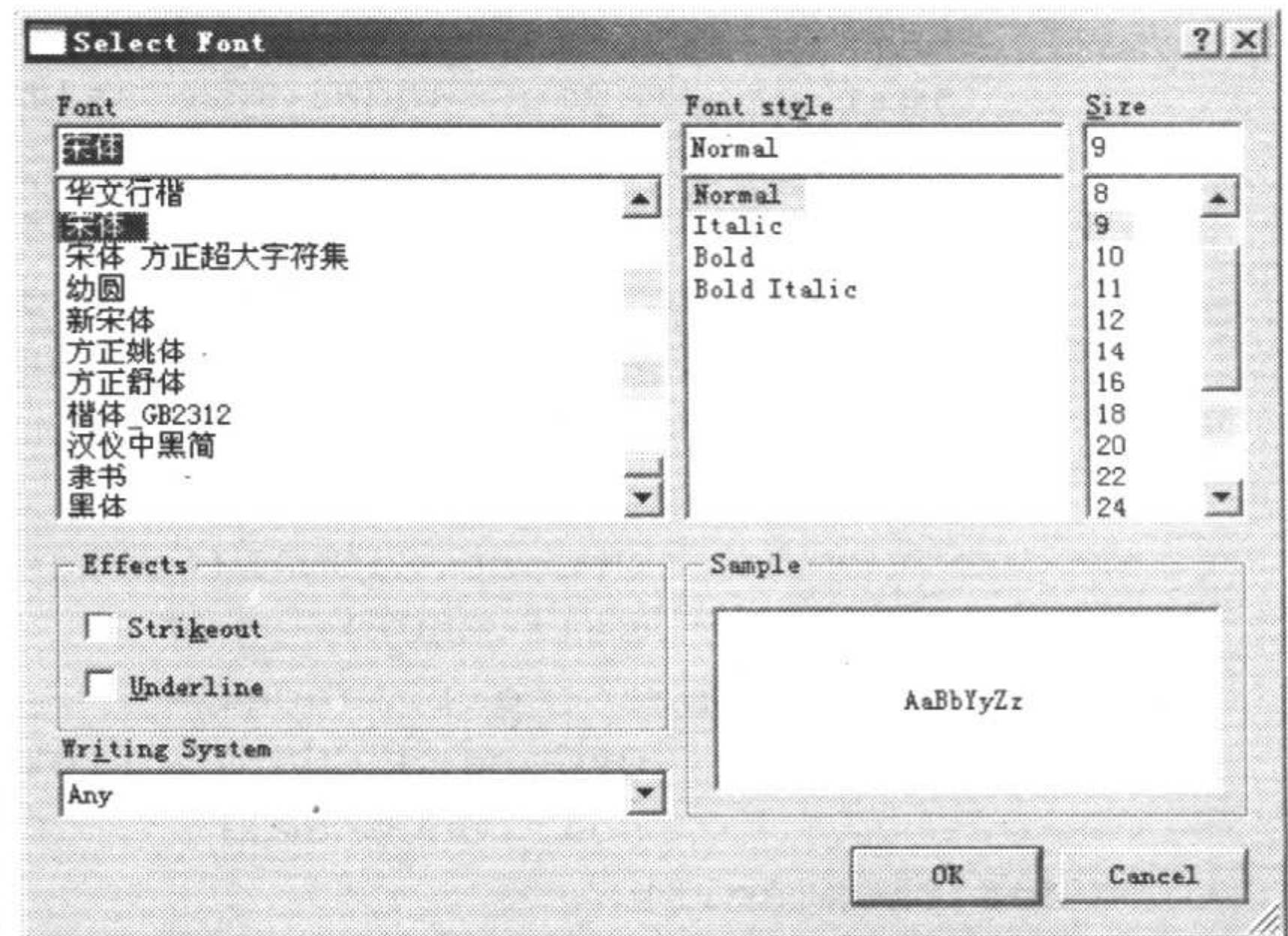


图 15-27 字体选择对话框

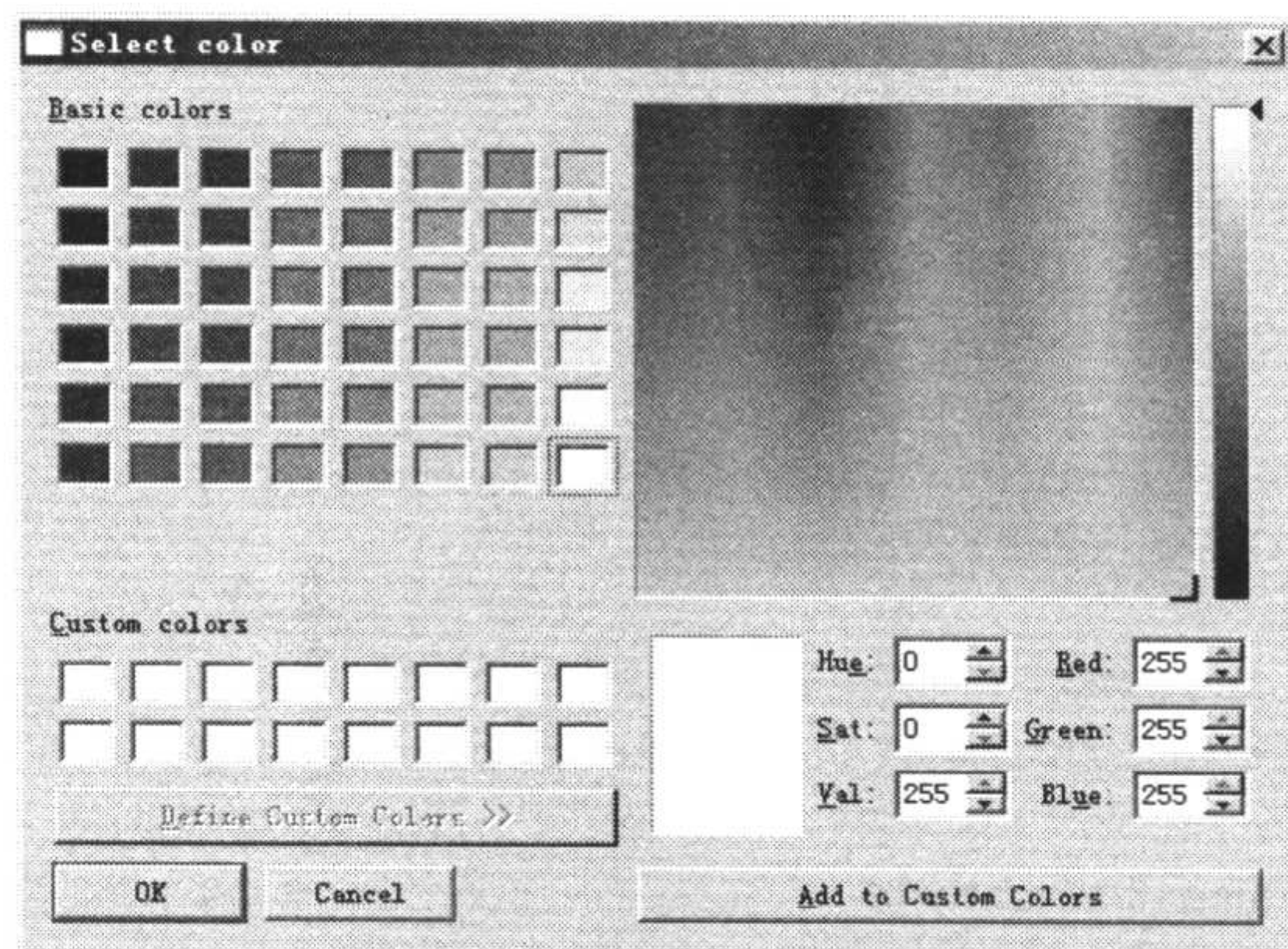


图 15-28 颜色选择对话框

15.3.2 自定义对话框

通过继承 `QtGui.QDialog` 类可以创建自定义的对话框。所创建的对话框和窗口一样，可以向其中添加组件。通过使用 `connect` 方法可以响应组件事件。如下所示的 `PyQtDialog.py` 使用 `QtGui.QDialog` 创建自定义对话框。

```
# -*- coding:utf-8 -*-
# file: PyQtDialog.py
#
import sys
from PyQt4 import QtCore, QtGui

class MyDialog(QtGui.QDialog):                                # 继承 QtGui.QDialog
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.gridlayout = QtGui.QGridLayout()                # 创建布局组件
        self.label = QtGui.QLabel('Input:')                  # 创建标签
        self.gridlayout.addWidget(self.label, 0, 0)
        self.edit = QtGui.QLineEdit()                         # 创建单行文本框
        self.gridlayout.addWidget(self.edit, 0, 1)
        self.ok = QtGui.QPushButton('Ok')                    # 创建 Ok 按钮
        self.gridlayout.addWidget(self.ok, 1, 0)
        self.cancel = QtGui.QPushButton('Cancel')            # 创建 Cancel 按钮
        self.gridlayout.addWidget(self.cancel, 1, 1)
        self.setLayout(self.gridlayout)
        self.connect(self.ok,                                # Ok 按钮事件
                      QtCore.SIGNAL('clicked()'),
                      self.OnOk)
        self.connect(self.cancel,                             # Cancel 按钮事件
                      QtCore.SIGNAL('clicked()'),
                      self.OnCancel)
    def OnOk(self):
        self.text = self.edit.text()                          # 处理 Ok 按钮事件
        self.done(1)                                           # 获取文本框中内容
                                                             # 结束对话框返回 1
```



```

def OnCancel(self):
    self.done(0)
class MyWindow(QtGui.QWidget):
    def __init__(self):
        QtGui.QWidget.__init__(self)
        self.setWindowTitle('PyQt')
        self.resize(300,200)
        gridlayout = QtGui.QGridLayout()
        self.button = QtGui.QPushButton('CreateDialog')
        gridlayout.addWidget(self.button, 1, 1)
        self.setLayout(gridlayout)
        self.connect(self.button,
                     QtCore.SIGNAL('clicked()'),
                     self.OnButton)
    def OnButton(self):
        dialog = MyDialog()
        r = dialog.exec_()
        if r:
            self.button.setText(dialog.text)
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()

```

处理 Cancel 按钮事件
结束对话框返回 0

初始化方法
调用父类初始化方法
设置窗口标题
设置窗口大小
创建布局组件
生成 Button1

向窗口中添加布局组件
Button 事件

处理按钮事件
创建对话框对象
运行对话框

运行 PyQtDialog.py 脚本后，单击 **【CreateDialog】** 将创建如图 15-29 所示的对话框。

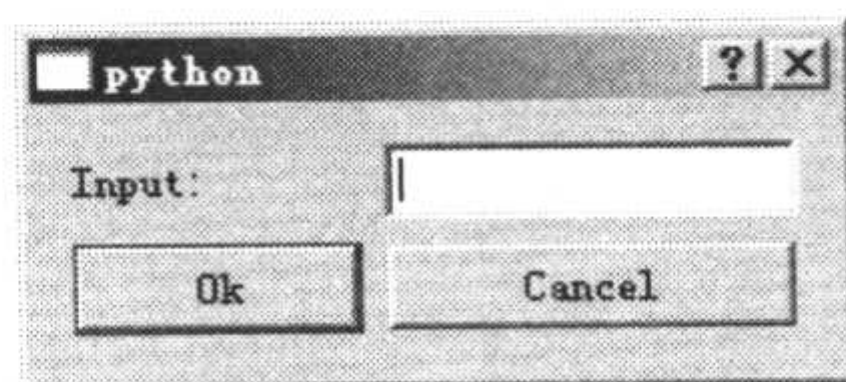


图 15-29 自定义对话框

15.4 资源文件

在 Qt 中资源文件是以“.ui”为后缀的文件。Qt 提供了 Qt Designer 用于创建资源文件。Qt Designer 是随 Qt 安装的，不必单独安装。使用 Qt Designer 创建的资源文件也可以在 PyQt 中使用。使用资源文件可以简化界面设计，也可以将界面和代码分离，提高程序的可维护性。

15.4.1 使用 Qt Designer 创建资源文件

Qt Designer 是所见即所得的资源文件编辑器，使用 Qt Designer 可以方便地创建复杂的 GUI 界面。使用 PyQt 中的 uic 模块可以在脚本中载入资源文件，创建 GUI 界面。使用 Qt Designer 创建资源文件的操作步骤如下所示。

(1) 单击 **【开始】** | **【所有程序】** | **【Qt by Trolltech v4.2.3 (OpenSource)】** | **【Designer】** 命令，运行 Qt Designer，如图 15-30 所示。

(2) 单击【Create】按钮创建一个对话框，如图 15-31 所示。重新调整对话框大小。

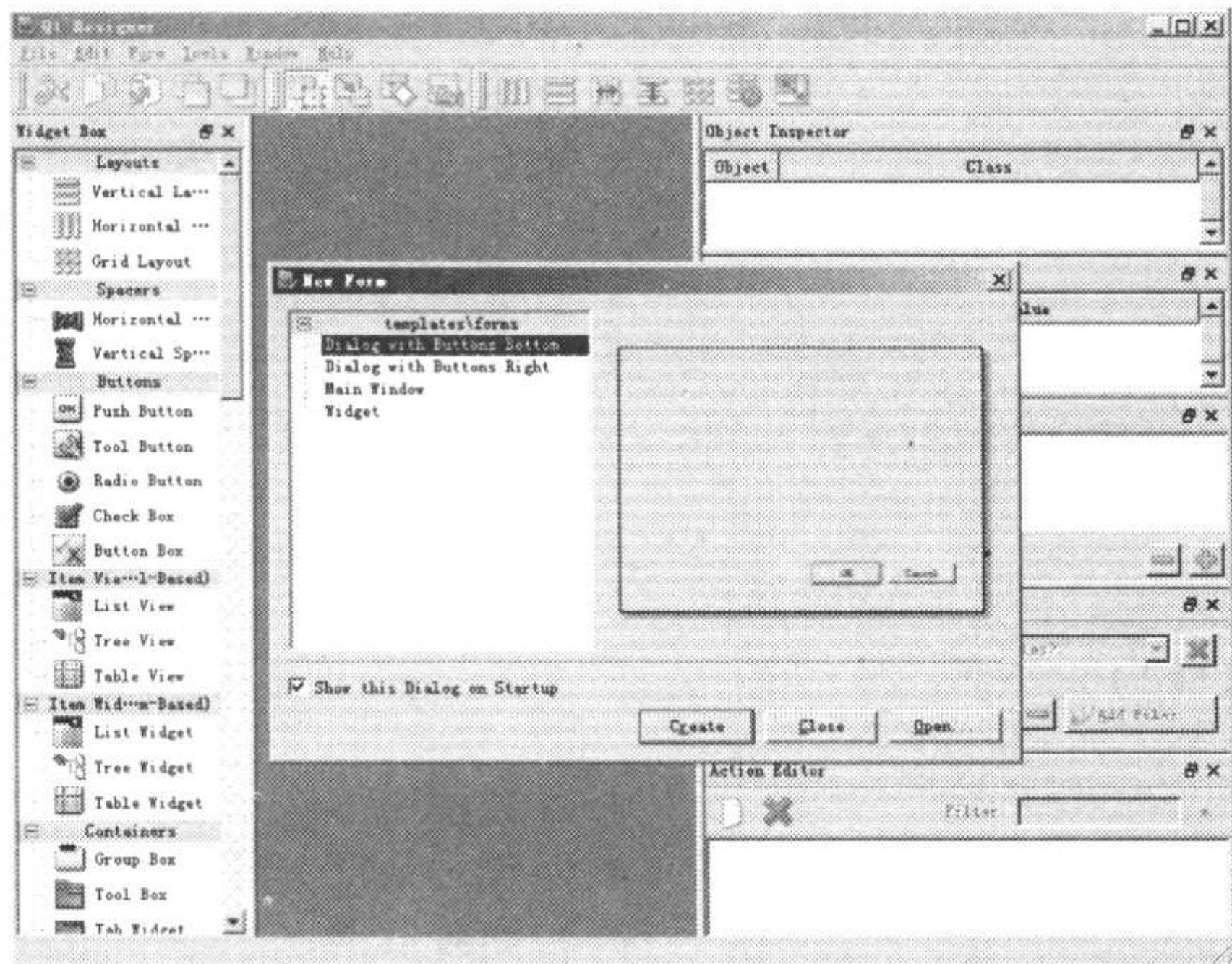


图 15-30 Qt Designer 窗口

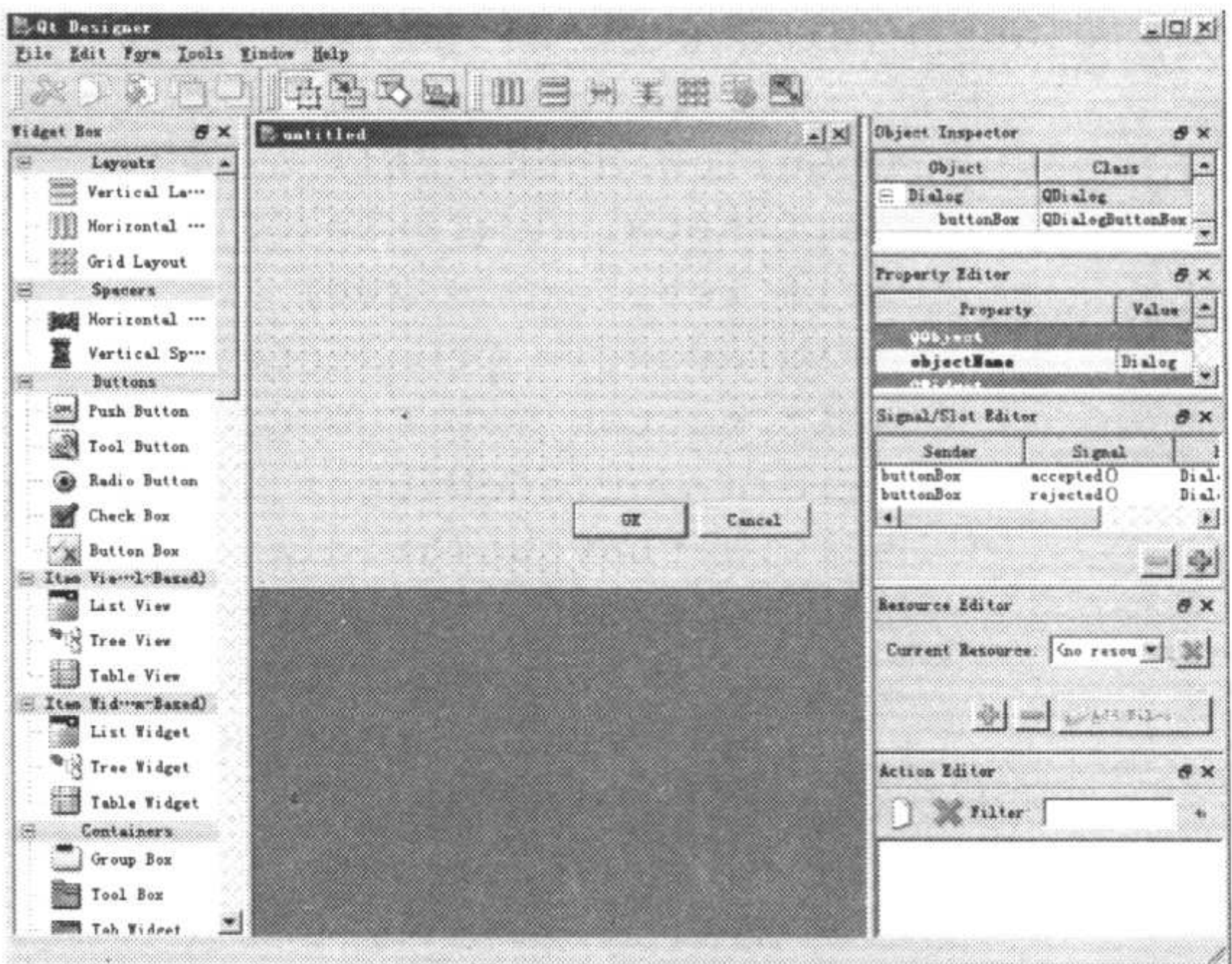


图 15-31 创建对话框

(3) 在【Widget Box】浮动窗口中的【Display Widgets】下的【Label】项上按住鼠标左键，将其拖放到所创建的对话框中，如图 15-32 所示。

(4) 选择【Property Editor】浮动窗口中的【text】项，将【Value】改为“Input”，修改标签文本，如图 15-33 所示。

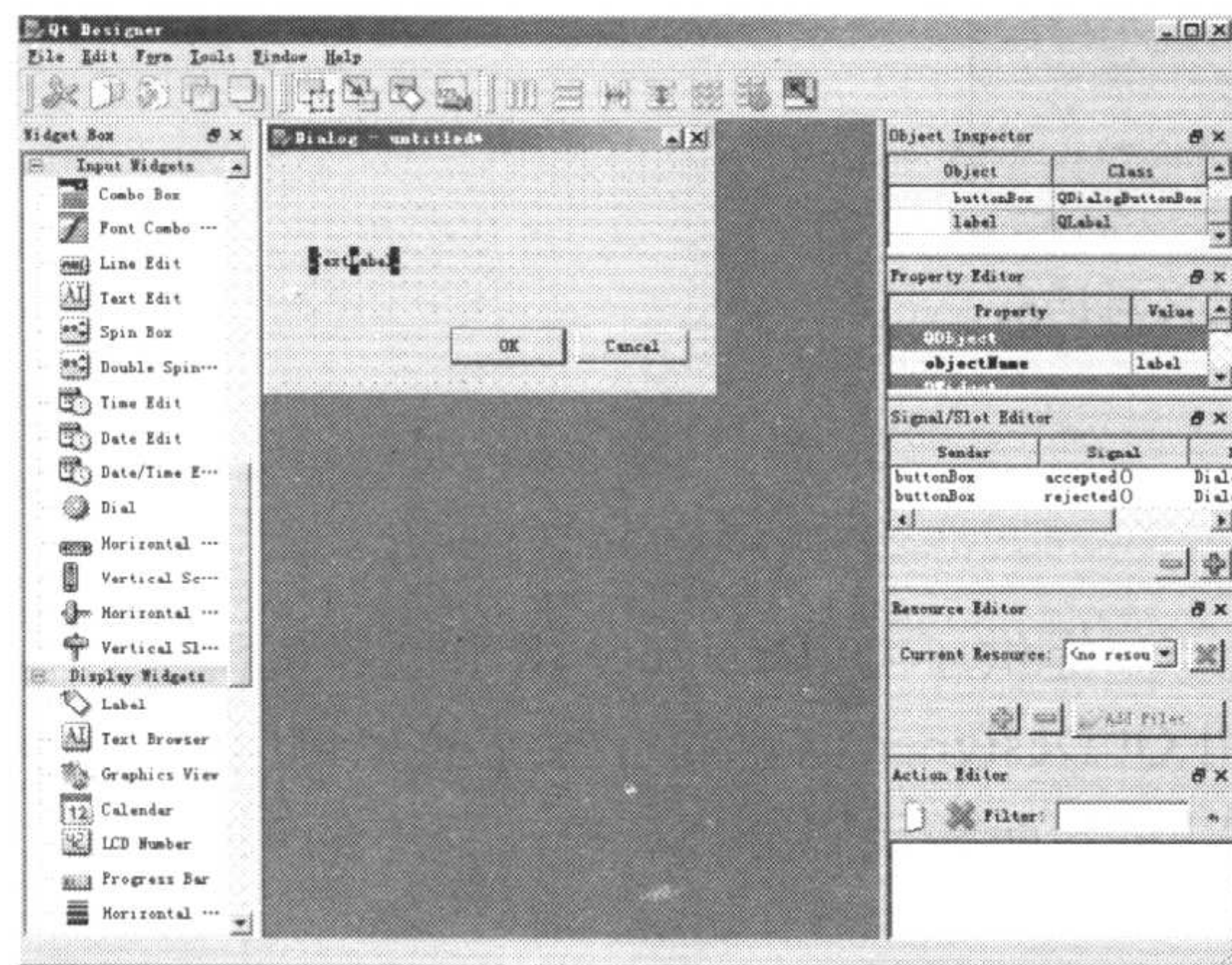


图 15-32 添加标签

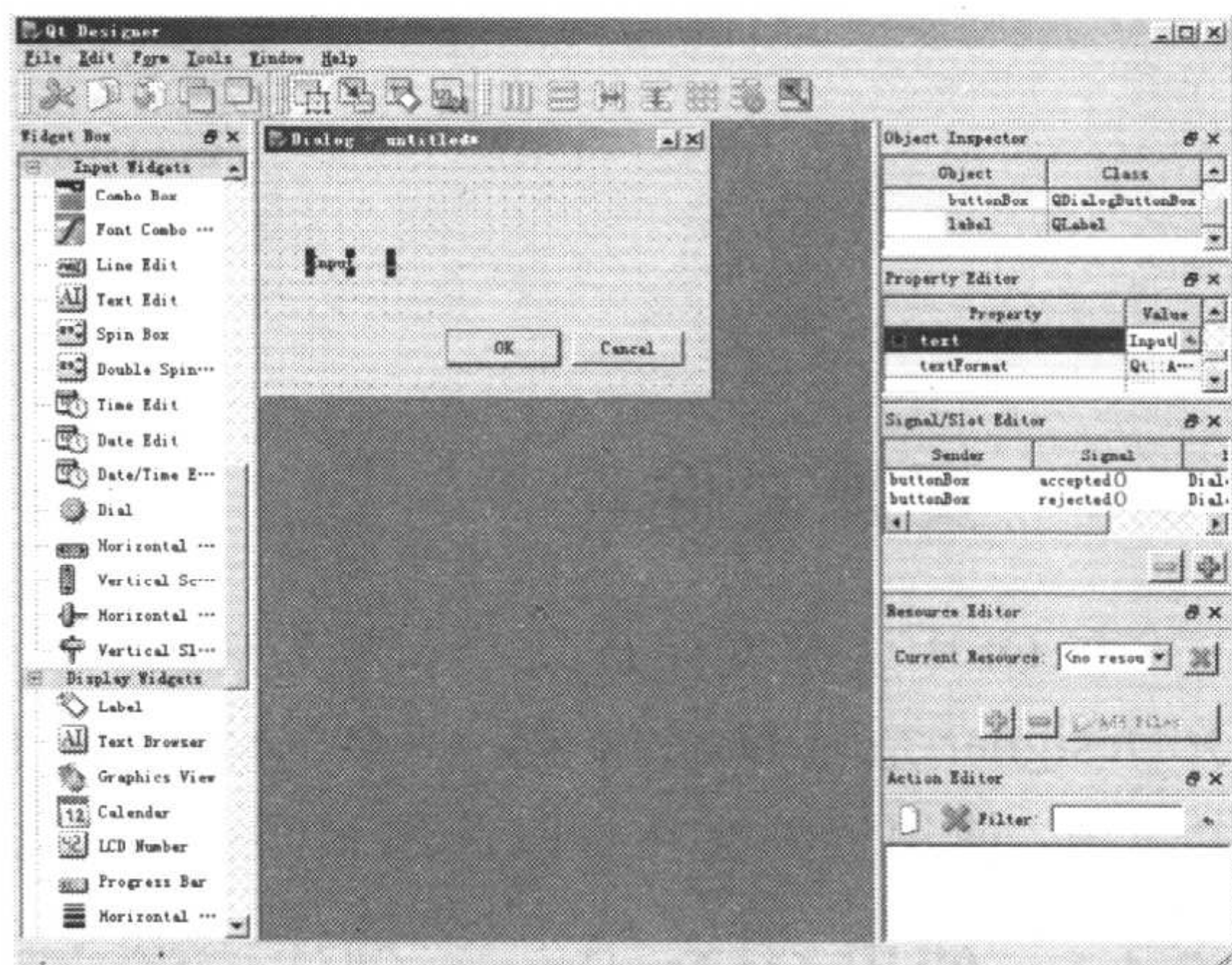


图 15-33 修改标签文本

(5) 在【Widget Box】浮动窗口中的【Input Widgets】下的【Line Edit】项上按住鼠标左键，将其拖放到所创建的对话框中，如图 15-34 所示。

(6) 选择【Property Editor】浮动窗口中的【Object Name】项，查看所添加的单行文本框的名字。改名字可以在脚本中使用。

(7) 单击 **【File】 | 【Save Form】** 命令，将资源文件保存为“res.ui”。

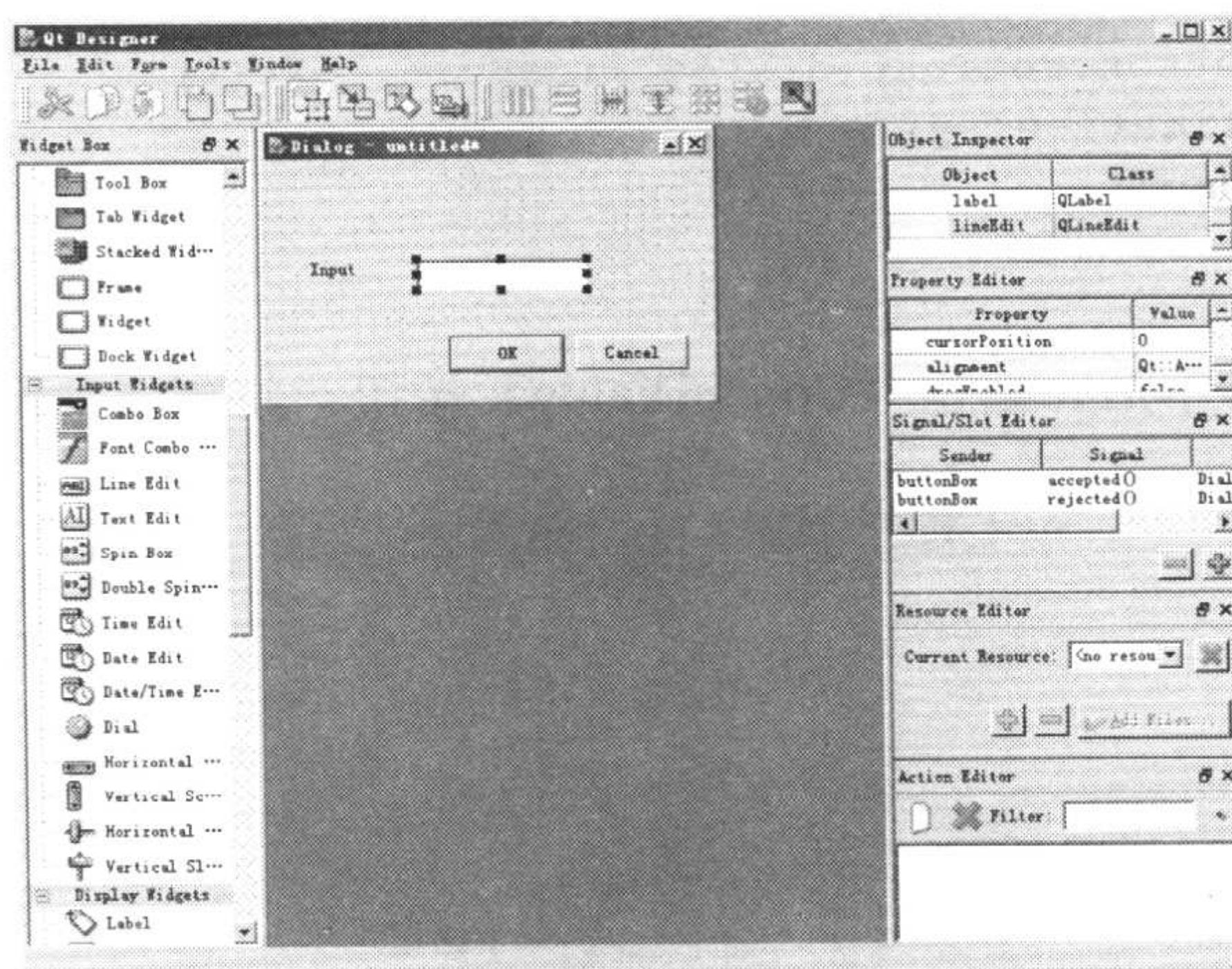


图 15-34 添加单行文本框

15.4.2 使用资源文件

使用 PyQt 中的 uic 模块可以载入资源文件中的组件。在脚本中只需使用 `uic.loadUi` 就可以载入资源文件。如下所示的 `PyQtRes.py` 脚本将 `PyQtDialog.py` 脚本改写，简化了对话框的创建。

```
# -*- coding:utf-8 -*-
# file: PyQtRes.py
#
import sys
from PyQt4 import QtCore, QtGui, uic

class MyDialog(QtGui.QDialog):                                # 继承 QtGui.QDialog
    def __init__(self):
        QtGui.QWidget.__init__(self)
        uic.loadUi("res.ui", self)                            # 载入资源文件

class MyWindow(QtGui.QWidget):
    def __init__(self):                                       # 初始化方法
        QtGui.QWidget.__init__(self)                        # 调用父类初始化方法
        self.setWindowTitle('PyQt')                        # 设置窗口标题
        self.resize(300,200)                                # 设置窗口大小
        gridlayout = QtGui.QGridLayout()                    # 创建布局组件
        self.button = QtGui.QPushButton('CreateDialog')     # 生成 Button1
        gridlayout.addWidget(self.button, 1, 1)
        self.setLayout(gridlayout)                          # 向窗口中添加布局组件
        self.connect(self.button,                           # Button 事件
            QtCore.SIGNAL('clicked()'),
            self.OnButton)

    def OnButton(self):                                     # 处理按钮事件
```



```
dialog = MyDialog()                                     # 创建对话框对象
r = dialog.exec_()                                       # 运行对话框
if r:
    self.button.setText(dialog.lineEdit.text())
app = QtGui.QApplication(sys.argv)
mywindow = MyWindow()
mywindow.show()
app.exec_()
```

运行 PyQtRes.py 脚本后，单击 **【CreateDialog】** 按钮，将创建如图 15-35 所示的对话框。

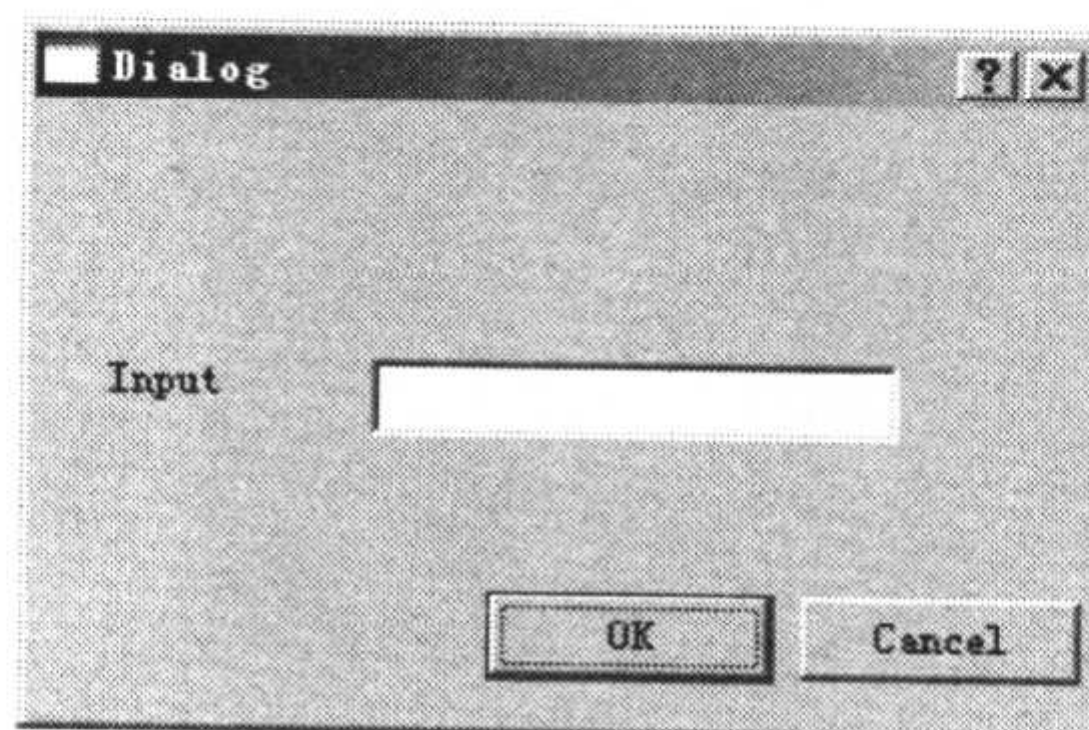
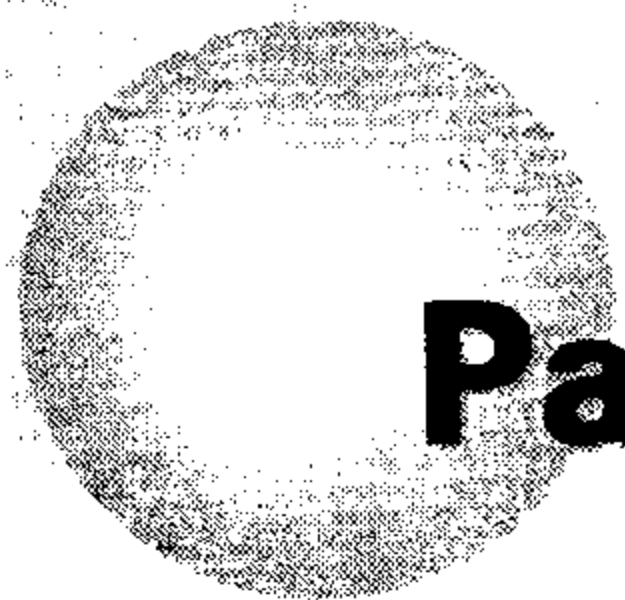


图 15-35 使用资源文件



Part 4

第四篇

Web 与数据库

第 16 章 Python 与数据库

数据库主要用于存储数据。数据库中的数据按照一定的模型进行组织和存储。Python 提供了对大多数数据库的支持。在 Python 中可以连接到数据库，进行查询、添加、删除数据等操作。

16.1 连接 Access 数据库


Access 数据库是 Microsoft Office 中的一部分。Access 数据库被称为桌面数据库，适用于中小型的应用系统中。Access 具有良好的界面、简单的操作，对于要求不高的应用，使用 Access 数据库是不错的选择。在 Python 中可以通过多种方式对 Access 数据库进行操作。

16.1.1 使用 ODBC 连接 Access 数据库

ODBC (Open Database Connectivity) 是 Microsoft 提出的数据库访问接口标准。ODBC 提供了独立于数据库的 API 函数，使用 ODBC 可以使用统一的方式访问不同的数据库。对于不同的数据库的支持是由 ODBC 的驱动层实现的。PythonWin 中的 `odbc` 模块提供了对 ODBC 的支持。PythonWin 中的 `dbi` 模块定义了各种数据类型。

1. 创建数据库

使用数据库之前应在 Access 中首先创建一个数据库，具体操作步骤如下所示。

- (1) 打开 Access，单击【文件】|【新建】命令，如图 16-1 所示。
- (2) 单击右侧新建文件浮动窗口中的【空数据库】项，将创建如图 16-2 所示的文件新建数据库对话框，将文件名改为“python”。
- (3) 单击【创建】按钮后，将打开如图 16-3 所示的窗口。
- (4) 双击【使用设计器创建表】项，将打开表设计窗口。向其中添加如图 16-4 所示的项。
- (5) 单击【文件】|【保存】命令，将弹出保存对话框，将表名命名为“people”，如图 16-5 所示。
- (6) 单击【确定】按钮后将弹出如图 16-6 所示的设置主键消息框。
- (7) 单击【是】按钮后，表设计窗口如图 16-7 所示，其中的 ID 项前多了个  标志，表

示其为主键。

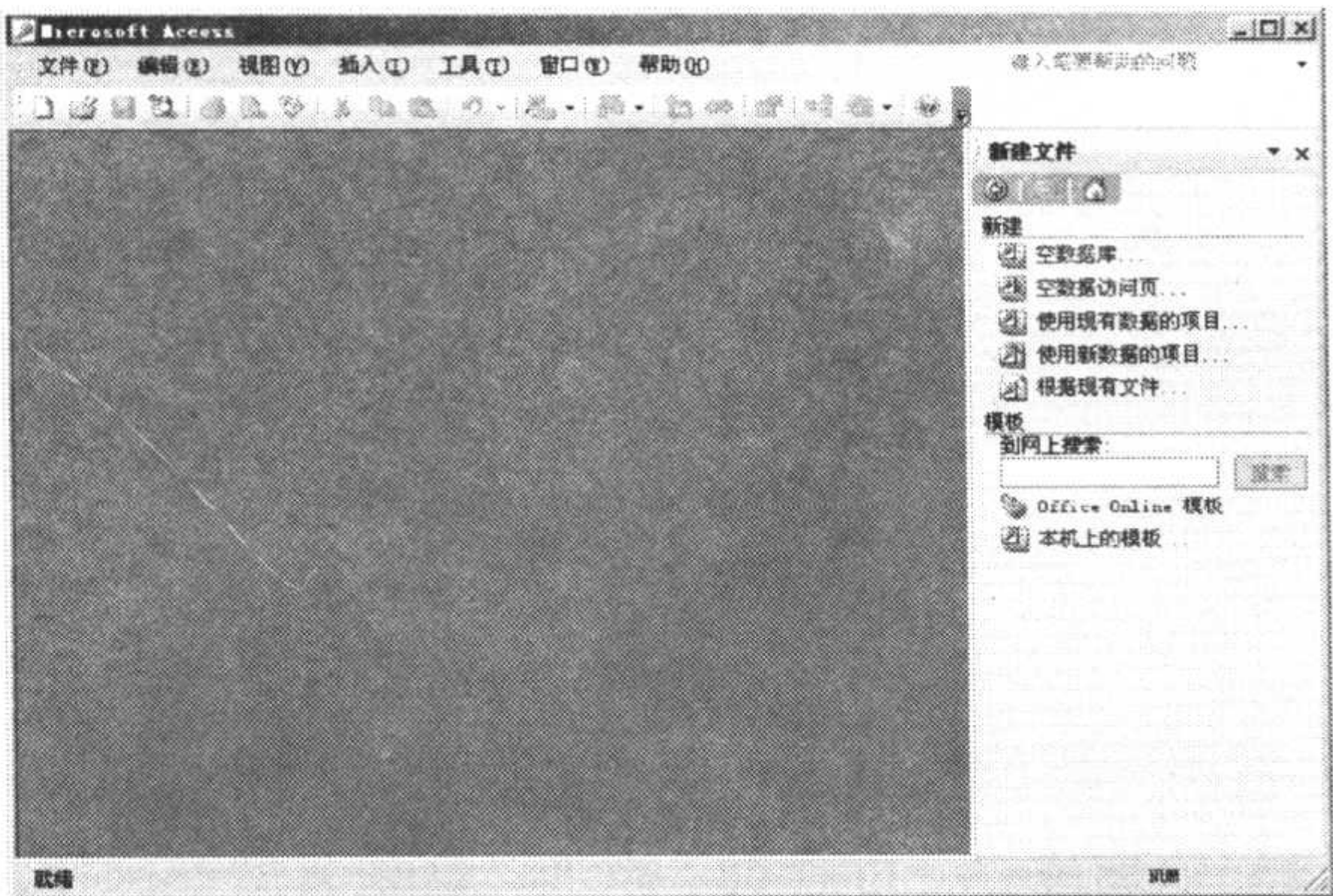


图 16-1 新建数据库

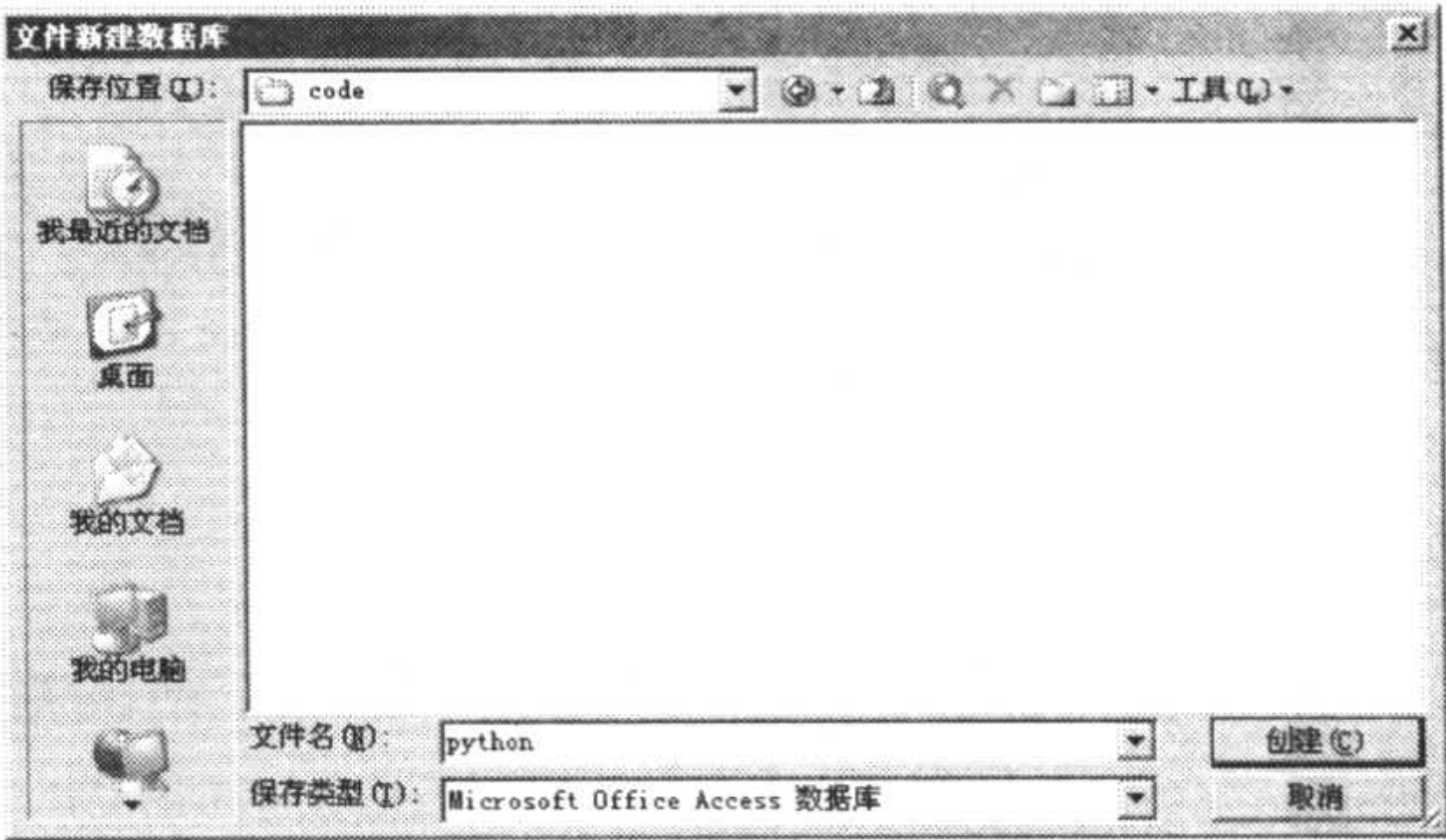


图 16-2 新建数据库对话框

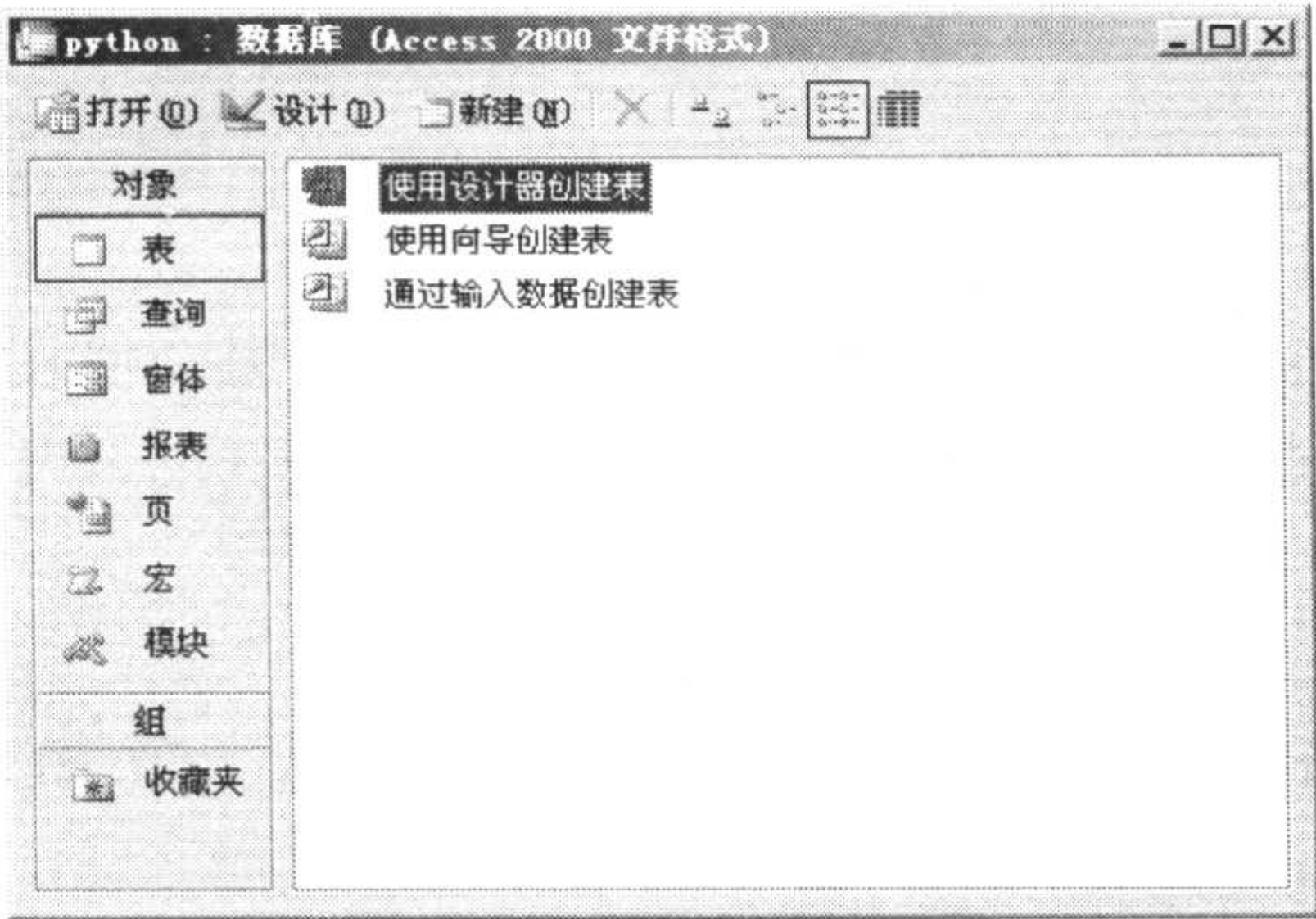


图 16-3 数据库操作窗口

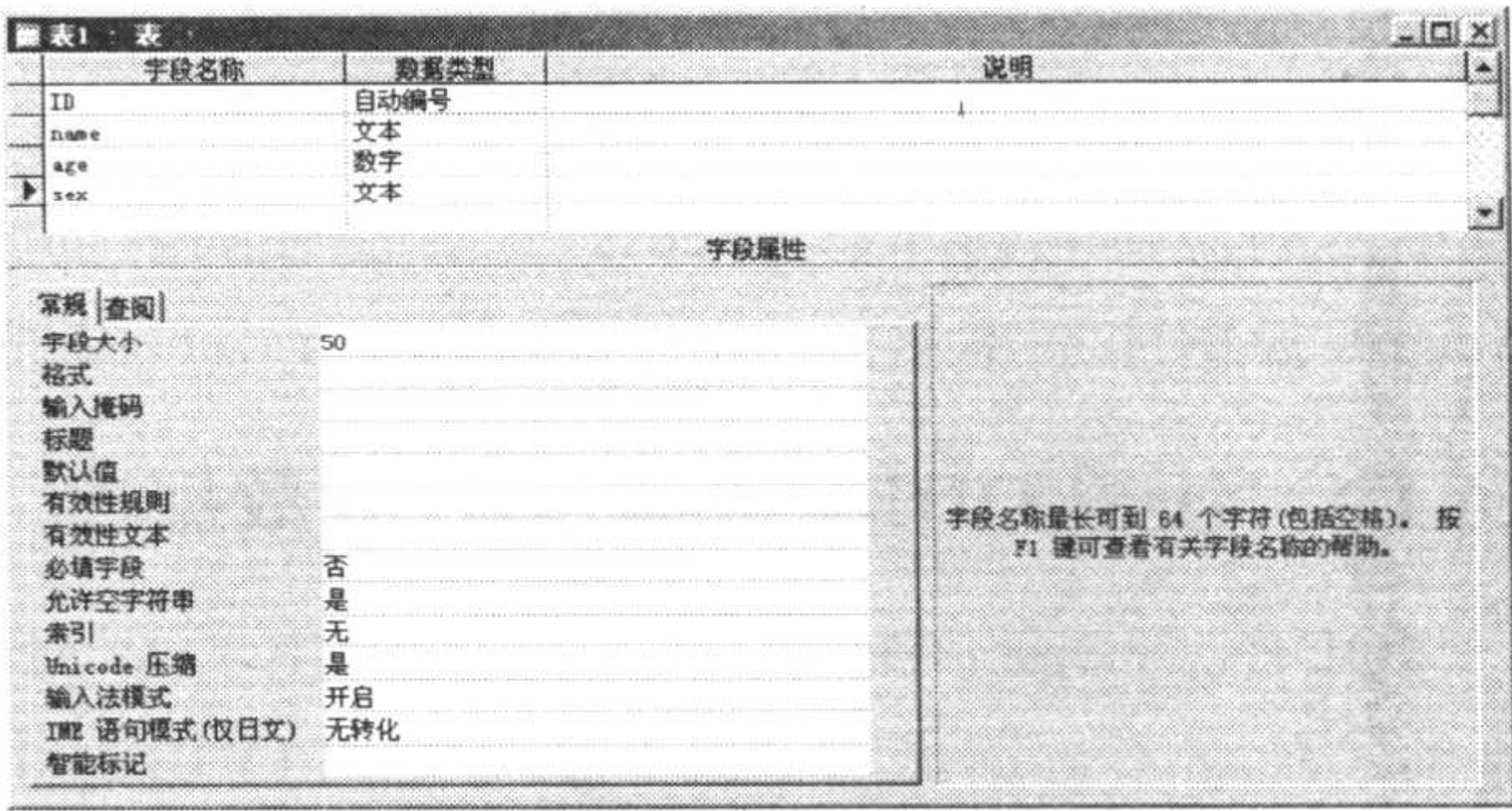


图 16-4 添加表项

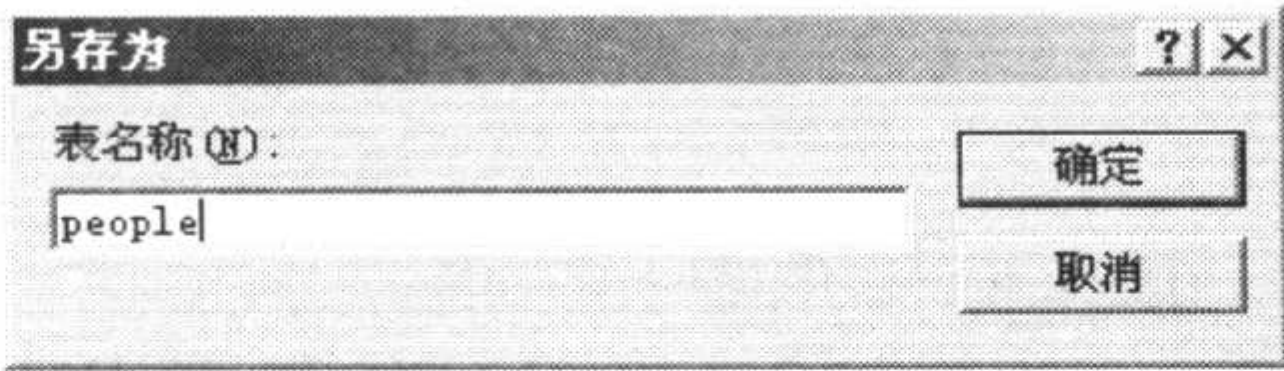


图 16-5 保存表

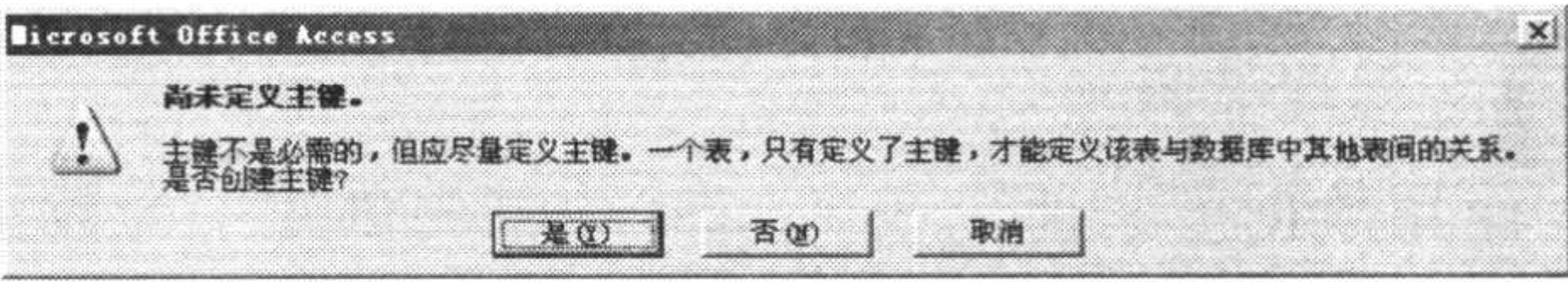


图 16-6 设置主键消息框

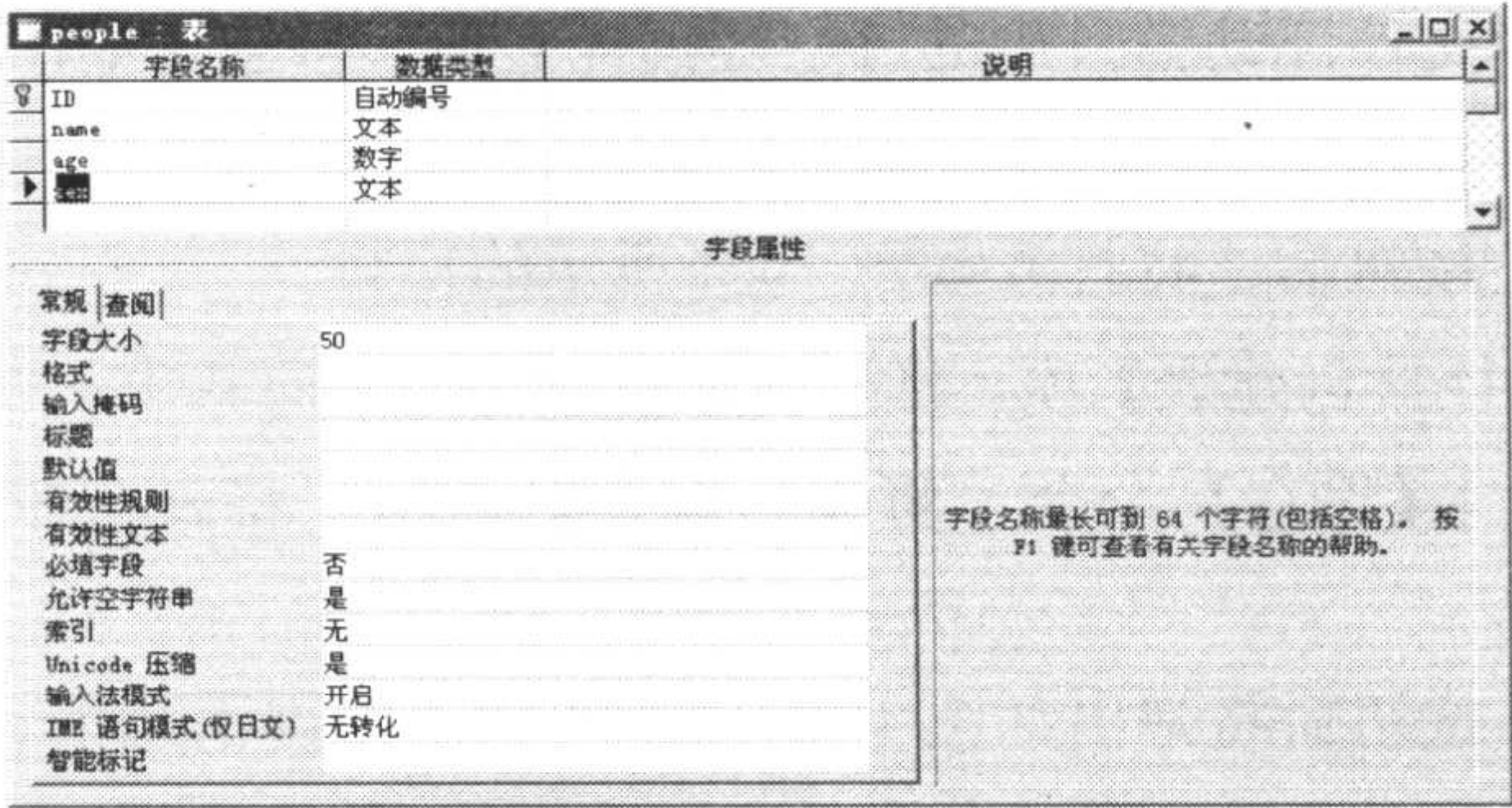


图 16-7 完成主键设置

第16章 Python 与数据库

(8) 关闭表设计窗口，双击图 16-8 所示窗口中的【people】项，将创建添加表内容窗口。

(9) 向其中添加如图 16-9 所示的项后，单击【文件】|【保存】命令，将创建的数据库保存。

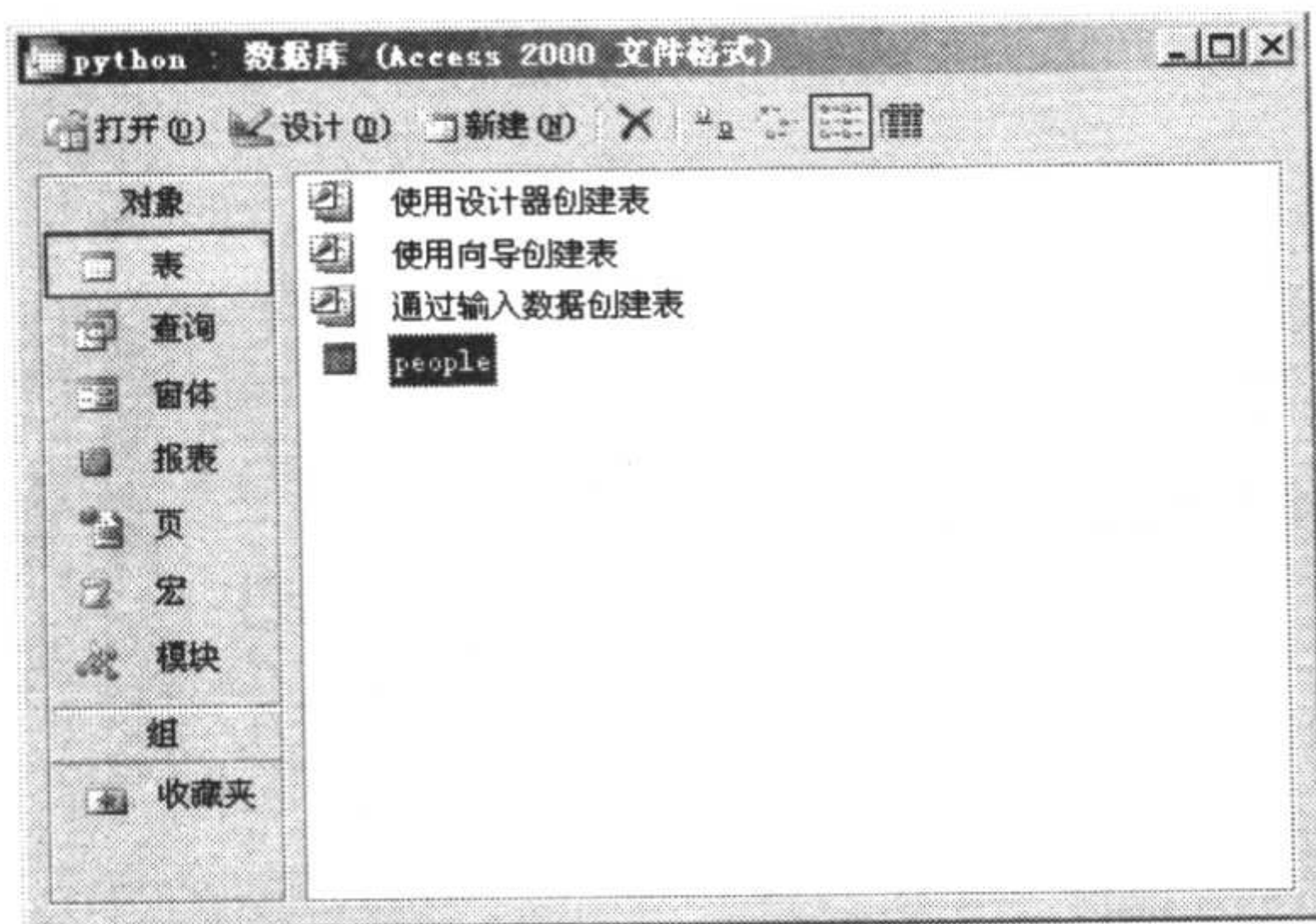


图 16-8 数据库操作窗口

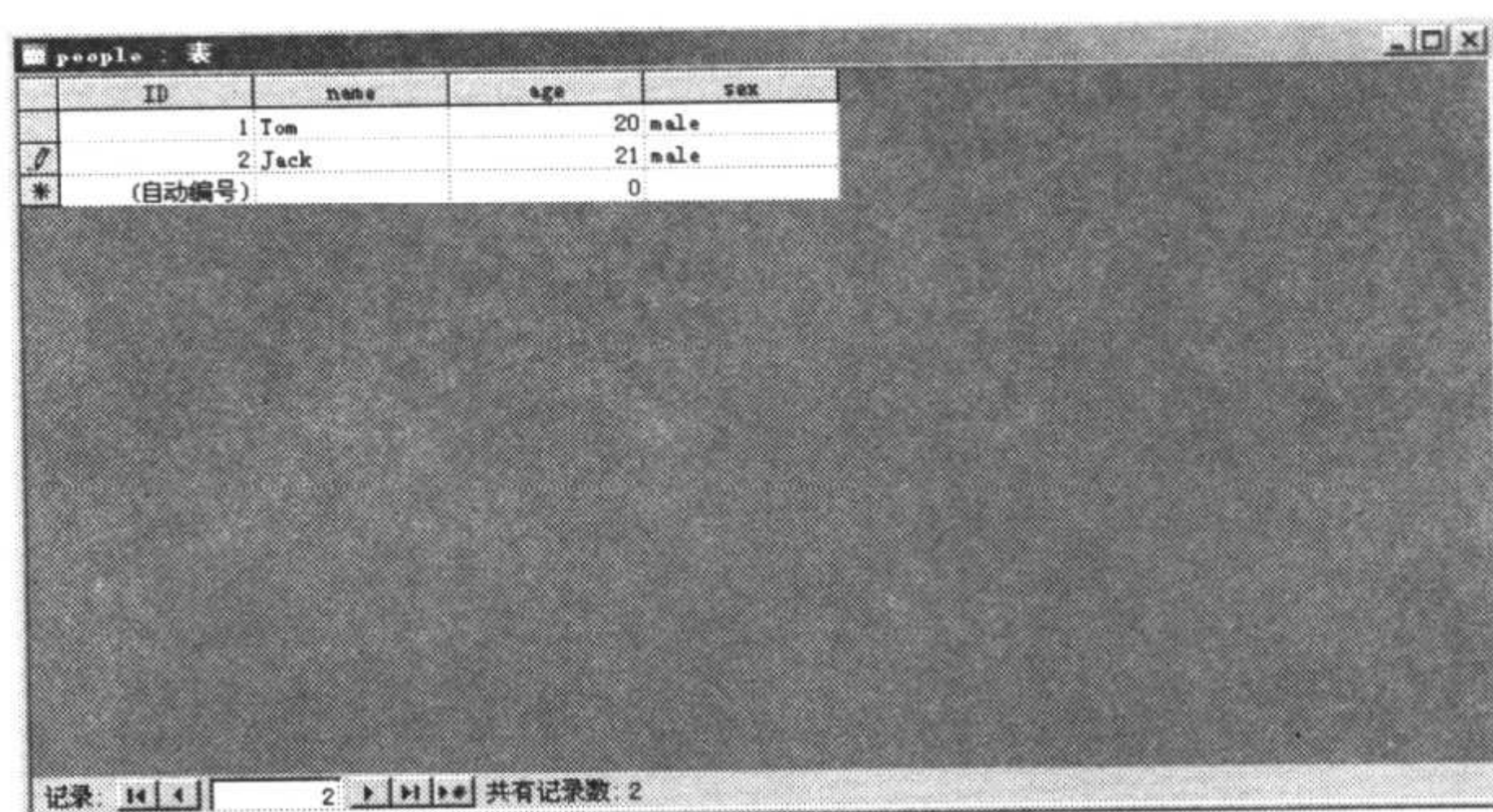


图 16-9 添加内容

2. 设置数据源

使用 ODBC 连接数据库时需要设置数据源，使其指向所创建的数据库。设置数据源的步骤如下所示。

(1) 单击【开始】|【控制面板】|【管理工具】|【数据源 (ODBC)】命令，如图 16-10 所示。

(2) 选择【用户 DSN】标签下【用户数据源】中的【MS Access Database】项，单击右侧的【添加】按钮，如图 16-11 所示。



图 16-10 ODBC 数据源管理器对话框



图 16-11 选择驱动

(3) 选择创建数据源对话框中的【Driver do Microsoft Access (*.mdb)】选项，单击【完

成】按钮后，将创建如图 16-12 所示的对话框。

(4) 在【数据源名】文本框中填入“podbc”，单击【选择】按钮，将弹出数据库路径选择对话框。选择上一小节中创建的数据库，如图 16-13 所示。

(5) 单击【确定】按钮，返回到如图 16-12 所示的对话框，单击【确定】按钮，完成设置。

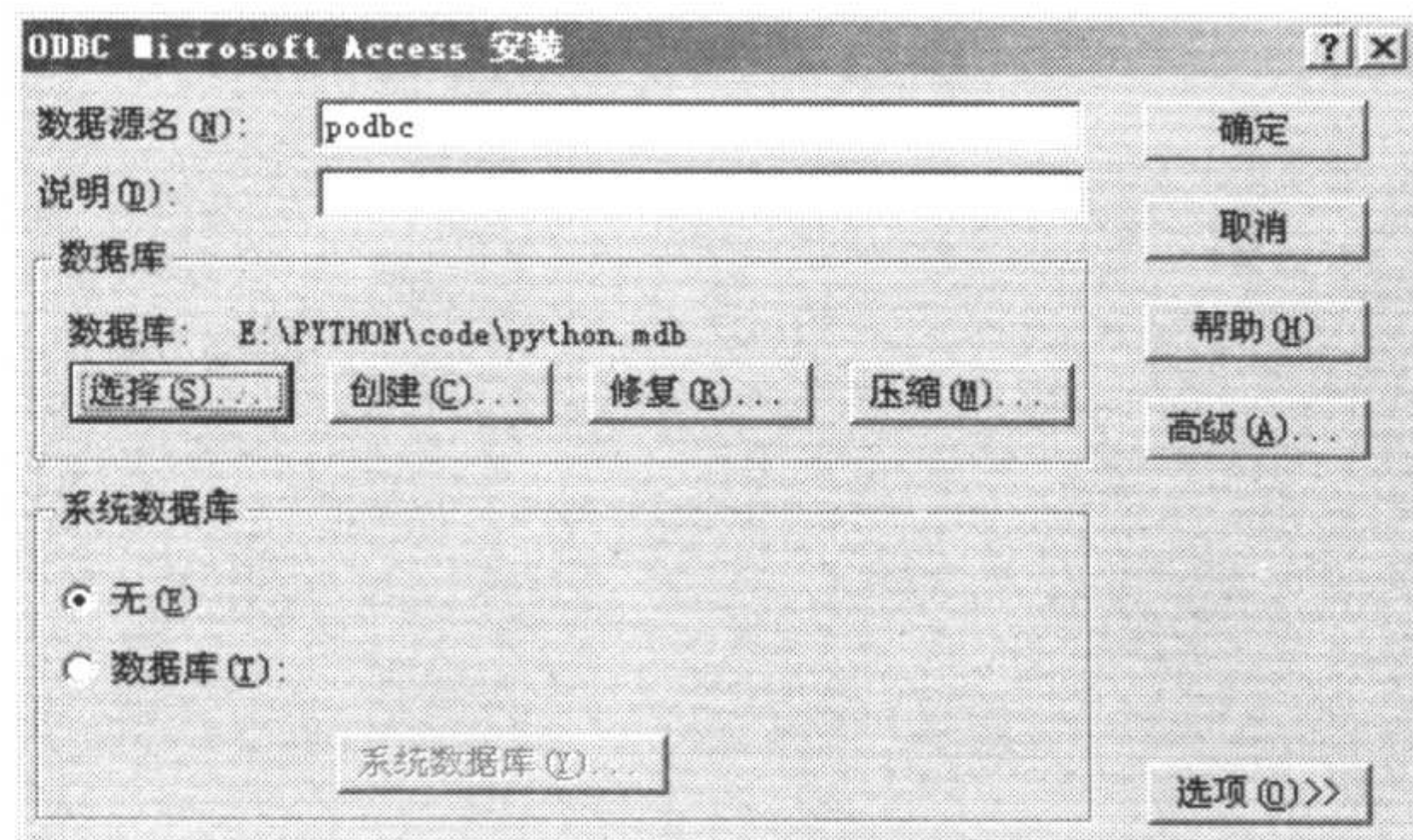


图 16-12 安装数据库

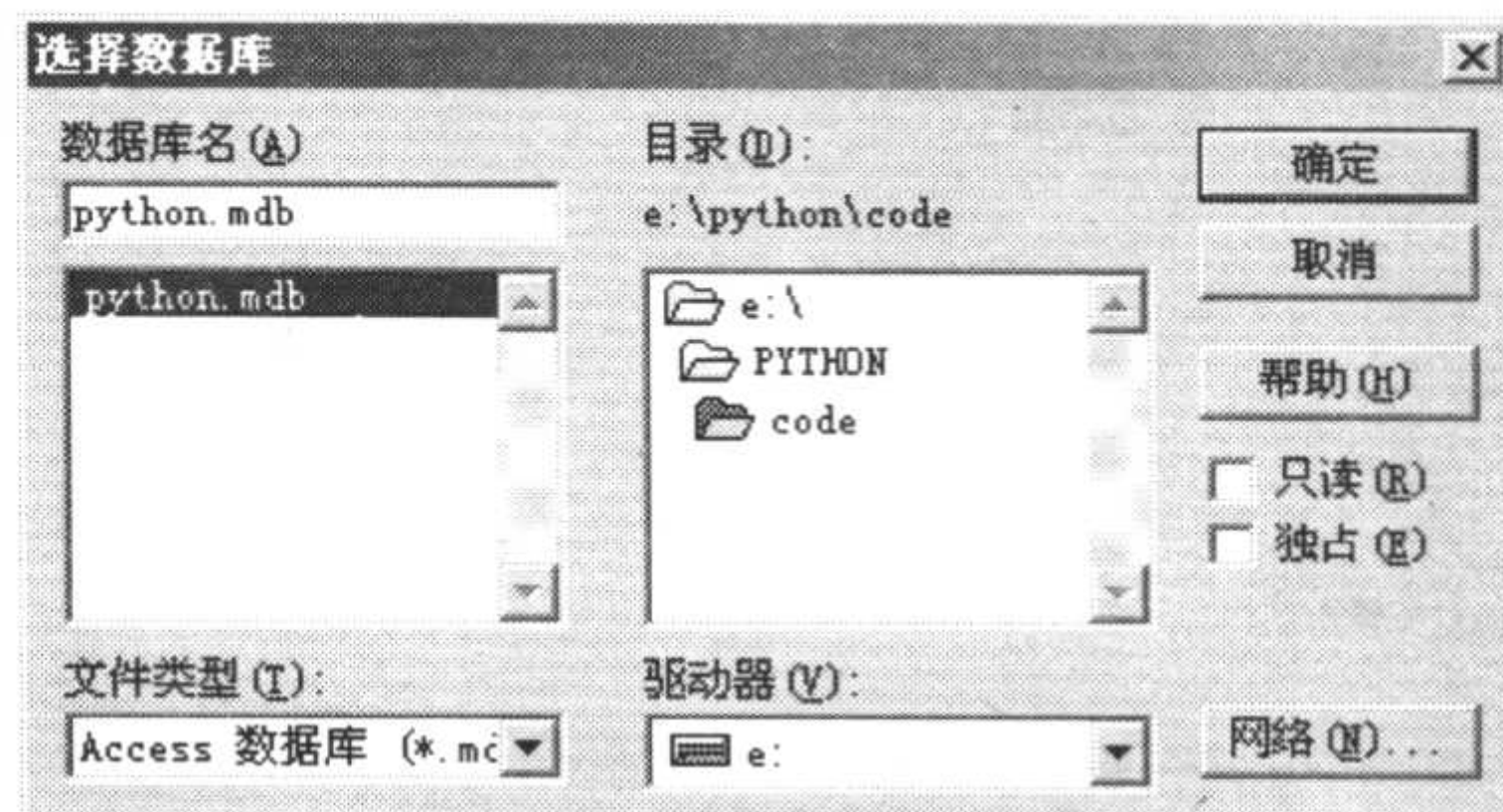


图 16-13 选择数据库路径

3. 连接数据库

完成数据源设置后就可以使用 PythonWin 中的 odbc 模块访问所添加的数据库。odbc 模块中仅提供了对数据的简单操作。使用 odbc 模块时应首先使用其 odbc 方法连接到数据库，创建一个 connection 对象。然后使用 connection 对象的 cursor 方法创建一个游标。使用游标对象的 execute 方法可以执行 SQL 语句，以对数据库进行操作。

当完成操作后应调用 cursor 的 close 方法关闭游标，然后调用 connection 对象的 close 方法关闭数据连接。如果执行了 SQL 查询语句，则可以使用 cursor 对象的 fetchall、fetchmany、fetchone 方法获取返回值。如下所示的 ODBC.py 脚本使用 PythonWin 的 odbc 模块对数据进行操作。

```
# -*- coding:utf-8 -*-
# file: ODBC.py
#
import odbc                                     # 导入 odbc 模块
con = odbc.odbc('podbc')                       # 连接到数据库，即在数据源名中填写的名字
cursor = con.cursor()                         # 创建 cursor 对象
cursor.execute('select id,name from people where id = 1')
                                                # 执行 SQL 语句查询 ID 为 1 的记录
r = cursor.fetchall()                         # 获得所有记录
print r                                       # 输出记录
cursor.execute('insert into people (name,age,sex)
               values (\'Jee\',21,\'female\')) # 添加记录
cursor.execute('DELETE FROM people where id = 3') # 删除 ID 为 3 的记录
con.commit()                                 # 提交事务
cursor.close()                              # 关闭 cursor
con.close()                                 # 关闭连接
```


16.1.2 使用 DAO 连接 Access 数据库

使用 ODBC 时需设置数据源，如果数据库的路径更改，相应地需重新修改数据源，这使得使用 ODBC 连接数据过于繁琐。对于一些数据简单应用的情况，可以使用 DAO(Data Access Objects) 代替 ODBC 连接数据库。

在 Python 中使用 DAO 时，需要使用 PythonWin 提供的 win32com 对象来使用 Windows 的 COM 组件。使用如下语句连接到 DAO 的 COM 对象上。

```
dbEngine = win32com.client.Dispatch('DAO.DBEngine.35')
```

然后通过 dbEngine 的 OpenDatabase 方法打开要连接的数据库。该方法返回一个数据连接，然后就可以使用 OpenRecordset 方法打开数据库中的表。该方法返回一个 Recordset 对象，使用它可以对数据库进行操作。Recordset 对象由 Field 对象组成，Field 对象即 Access 数据库中的一列。Recordset 对象的 Fields 表示了其所有 Field 对象的集合。Recordset 对象常用的方法有以下几种。

- AddNew: 添加新记录。
- Close: 关闭 Recordset 对象。
- Delete: 删除记录。
- Find: 查找记录。
- Move: 移动位置。
- MoveFirst: 移动到第一条记录。
- MoveLast: 移动到最后一条记录。
- MoveNext: 移动到下一条记录。
- MovePrevious: 移动到上一条记录。
- Update: 更新记录。

如果 Access 数据库使用较新版本的 Access 创建，则在使用 DAO 连接 Access 数据库时，应首先将数据库转换。使用 Access 打开数据库后，单击【工具】|【数据库实用工具】|【转换数据库】|【转为 Access 97 文件格式】命令，将数据库保存。如下所示的 DAO.py 脚本使用 DAO 连接 Access 数据库。

```
# -*- coding:utf-8 -*-
# file: DAO.py
#
import win32com.client
dbEngine = win32com.client.Dispatch('DAO.DBEngine.35')
daoDB = dbEngine.OpenDatabase('python.mdb')
daoRS = daoDB.OpenRecordset('people')
daoRS.MoveLast()
print daoRS.RecordCount
```

导入 win32com.client
连接 COM 对象
打开数据库
打开表
移动到最后一条记录
输出记录总数


```

print daoRS.Fields('name').Value      # 输出最后一条记录的 name
print daoRS.Fields('age').Value        # 输出最后一条记录的 age
print daoRS.Fields('sex').Value        # 输出最后一条记录的 sex
daoRS.AddNew()                         # 添加新记录
daoRS.Fields('name').Value = 'Kate'    # 新记录的 name
daoRS.Fields('age').Value = 22         # 新记录的 age
daoRS.Fields('sex').Value = 'Female'   # 新记录的 sex
daoRS.Update()                         # 更新记录
daoRS.Close()                         # 关闭表
daoDB.Close()                         # 关闭数据库连接

```

16.1.3 使用 ADO 连接 Access 数据库

ADO (ActiveX Data Objects) 是另一种简单的数据访问接口。使用 ADO 可以更快地创建连接数据库的应用程序，和 DAO 相比有很多相似之处，但使用 ADO 也需要设置数据源。ADO 中的 Recordset 对象常用的方法如下所示。

- AddNew: 添加新记录。
- Close: 关闭 Recordset 对象。
- Delete: 删除记录。
- Find: 查找记录。
- Move: 移动位置。
- MoveFirst: 移动到第一条记录。
- MoveLast: 移动到最后一条记录。
- MoveNext: 移动到下一条记录。
- MovePrevious: 移动到上一条记录。
- Update: 更新记录。

如下所示的 ADO.py 使用 ADO 连接 Access 数据库。

```

# -*- coding:utf-8 -*-
# file: ADO.py
#
import win32com.client
adoCon = win32com.client.Dispatch('ADODB.Connection')
adoCon.Open('podbc')
adoRS = win32com.client.Dispatch('ADODB.Recordset')
adoRS.Open('[' + 'people' + ']', adoCon, 1, 3)
adoRS.MoveFirst()
for i in range(adoRS.RecordCount):
    print adoRS.Fields('name').Value
    print adoRS.Fields('age').Value
    print adoRS.Fields('sex').Value
    adoRS.MoveNext()
adoRS.AddNew()

```

导入 win32com.client
创建连接对象
连接到数据源
创建 Recordset 对象
打开数据源中的 people 表
移动到第一条记录

输出记录的 name
输出记录的 age
输出记录的 sex

添加新记录


```

adoRS.Fields('name').Value = 'Kate'           # 新记录的 name
adoRS.Fields('age').Value = 22                 # 新记录的 age
adoRS.Fields('sex').Value = 'Female'          # 新记录的 sex
adoRS.Update()                                # 更新记录
adoRS.Close()                                 # 关闭表
adoCon.Close()                                # 关闭数据库连接

```

16.2 使用 MySQL 数据库

MySQL 是一个小巧的多用户、多线程 SQL 数据库服务器。MySQL 是一个客户机/服务器结构的实现，它由一个服务器守护进程和客户程序组成。MySQL 提供了对 SQL 语句的支持。在 Python 中可以使用 MySQLdb 模块连接到 MySQL，对 MySQL 数据库进行操作。

16.2.1 安装 MySQL

MySQL 的官方网站 <http://www.mysql.org> 提供了 Windows 版的 MySQL 安装程序。MySQL 的安装步骤较多，以 mysql-5.0.37-win32 为例，安装过程如下所示。

(1) 从 MySQL 官方网站下载 Windows (x86) ZIP/Setup.EXE 项，将下载的 mysql-5.0.37-win32.zip 文件解压，运行其中的 Setup.exe，如图 16-14 所示。

(2) 单击【Next】按钮，进入安装形式选择界面，如图 16-15 所示。其中“Typical”项为典型安装，将 MySQL 安装在 C 盘。“Complete”项为完整安装，将安装所有功能。“Custom”项为自定义安装，可以选择安装路径和所安装的功能。

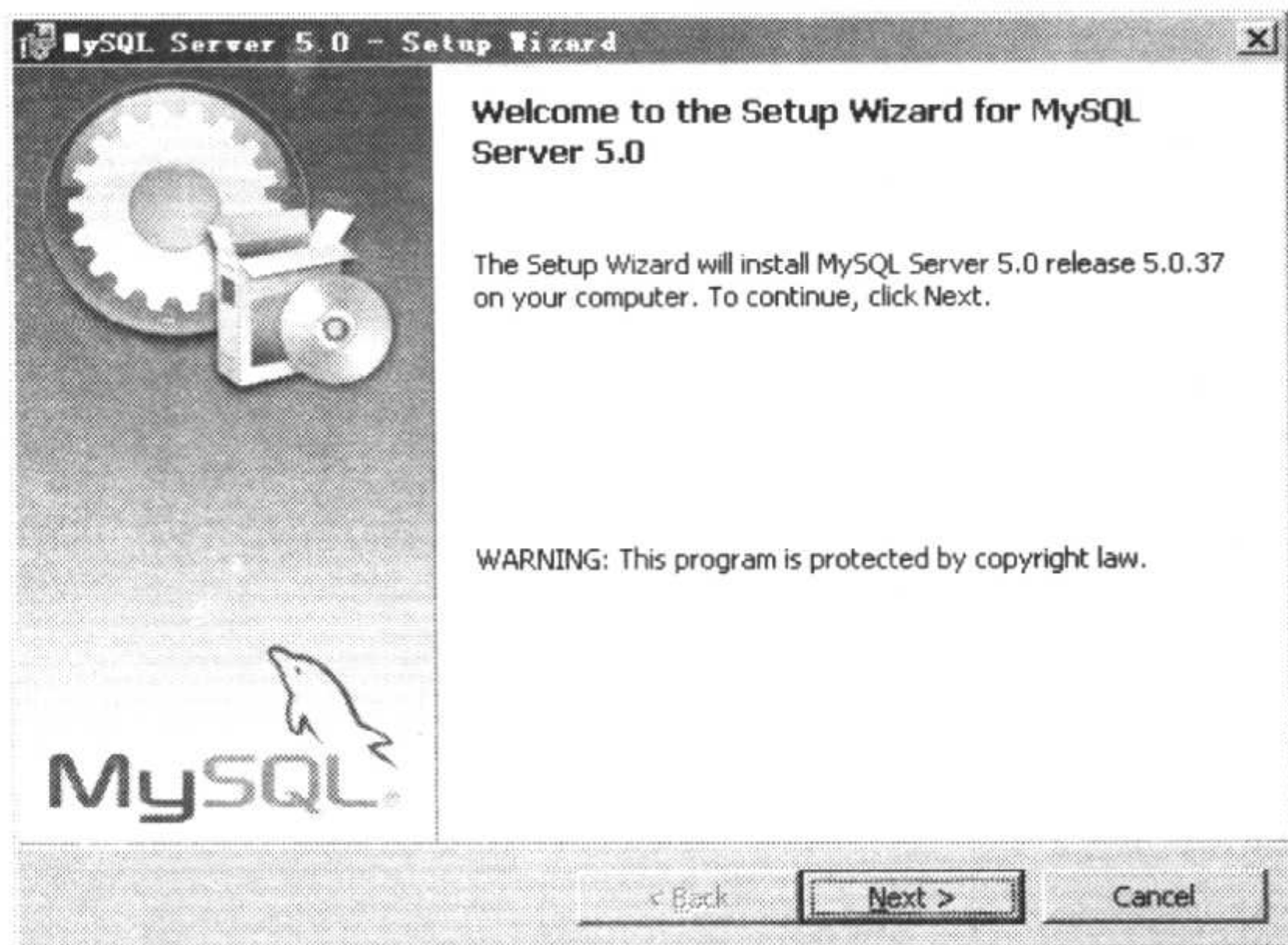


图 16-14 MySQL 安装

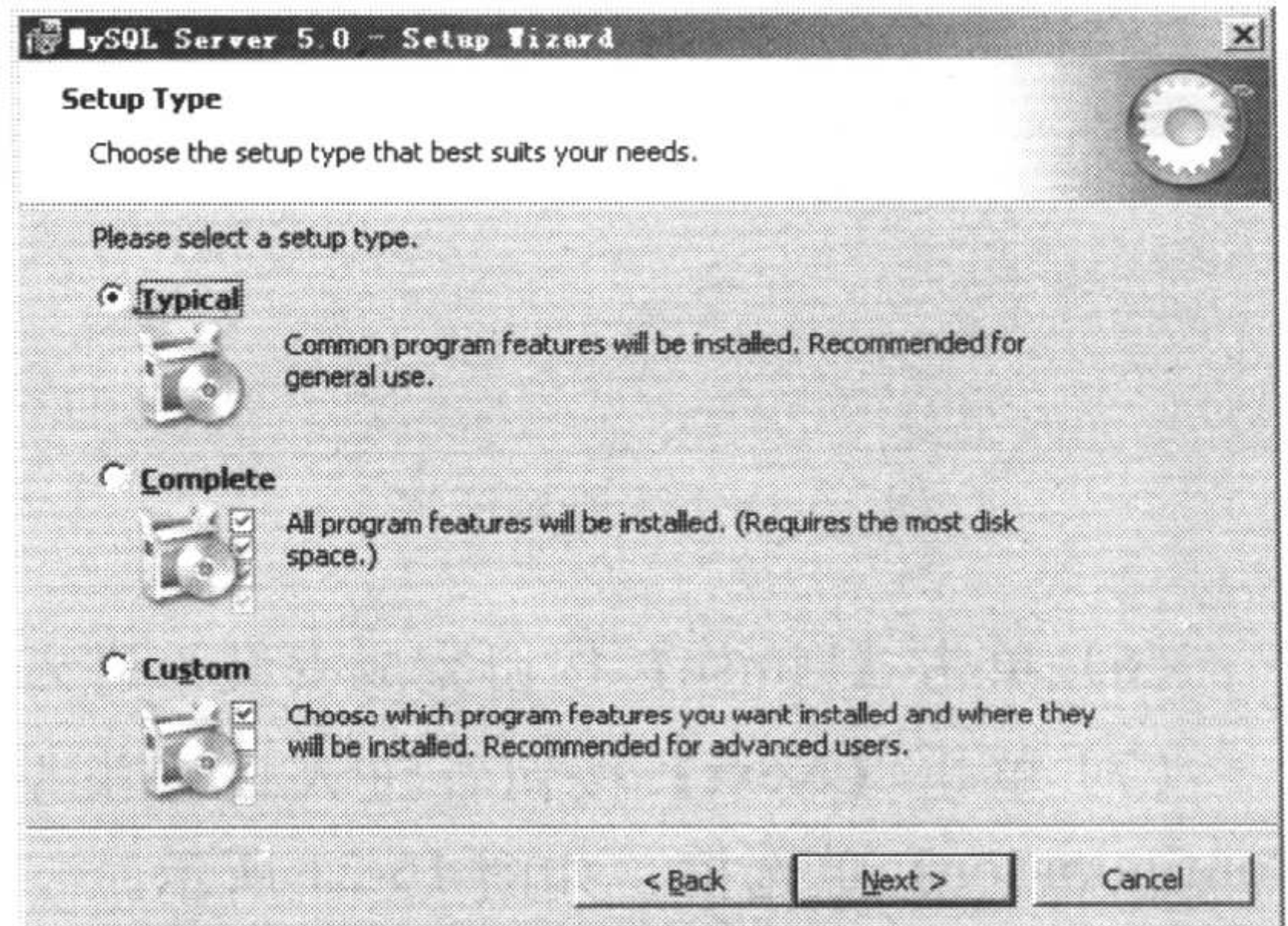


图 16-15 选择安装形式

(3) 如果需要重新选择安装路径，则需选中【Custom】单选框，单击【Next】按钮，进入下一步安装，如图 16-16 所示。单击【Change】按钮更改安装路径。

(4) 单击【Next】按钮后将出现如图 16-17 所示的确认安装对话框，单击【Install】按钮进行安装。

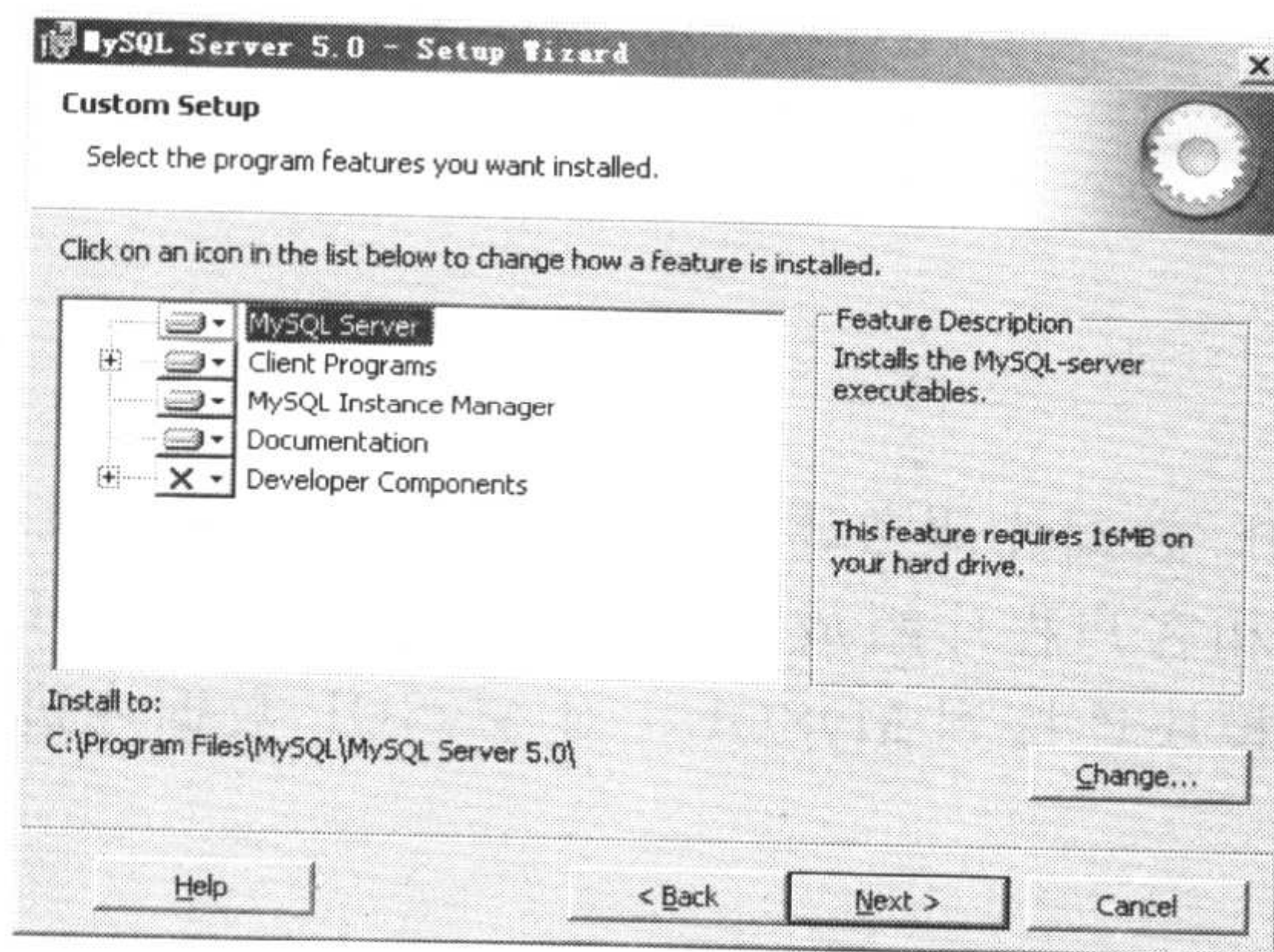


图 16-16 选择路径和功能

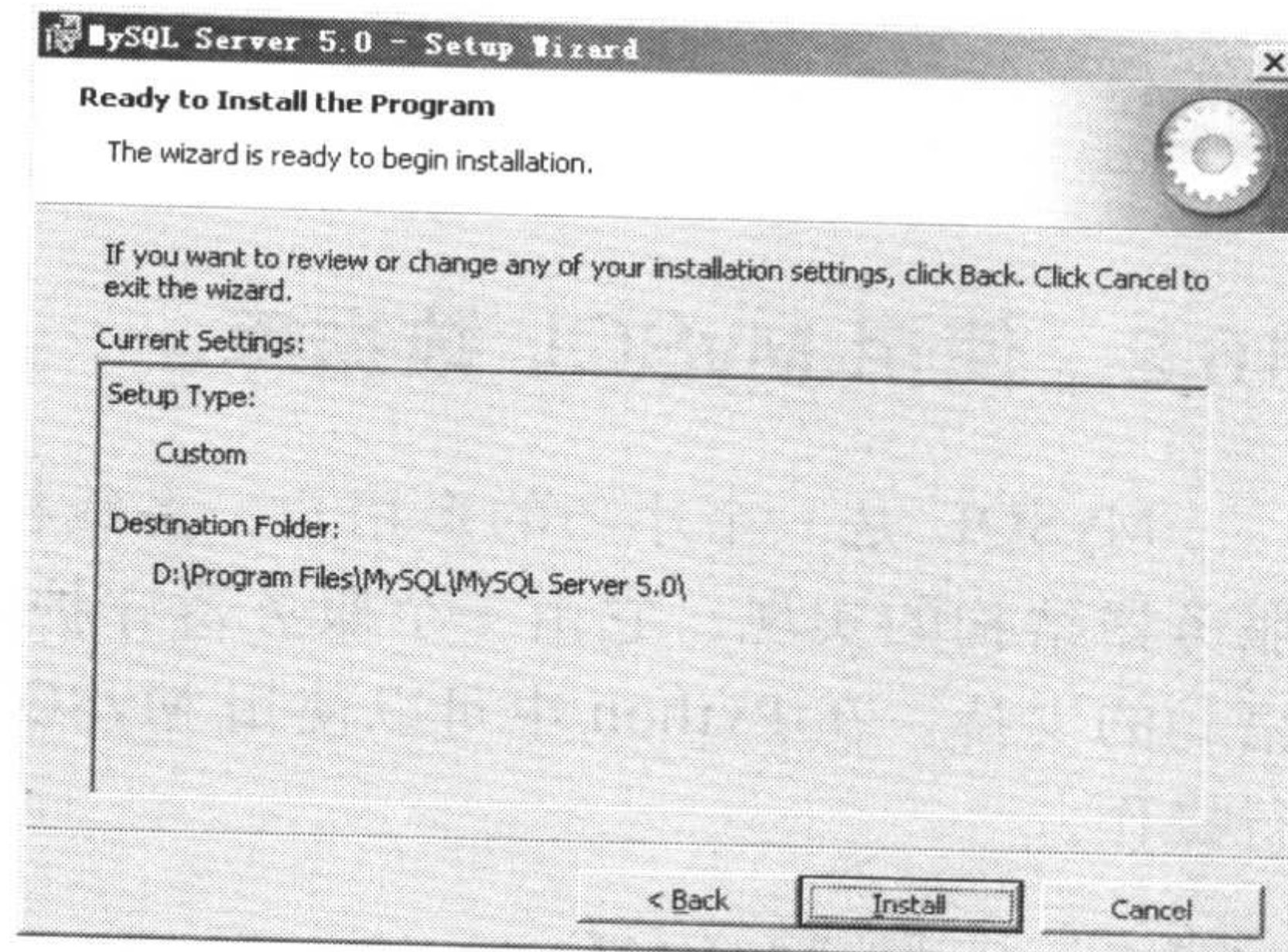


图 16-17 确认安装

(5) 文件复制完成后，将出现如图 16-18 所示的注册对话框，可以选中【Skip Sign-Up】单选框，单击【Next】按钮完成安装，如图 16-19 所示。

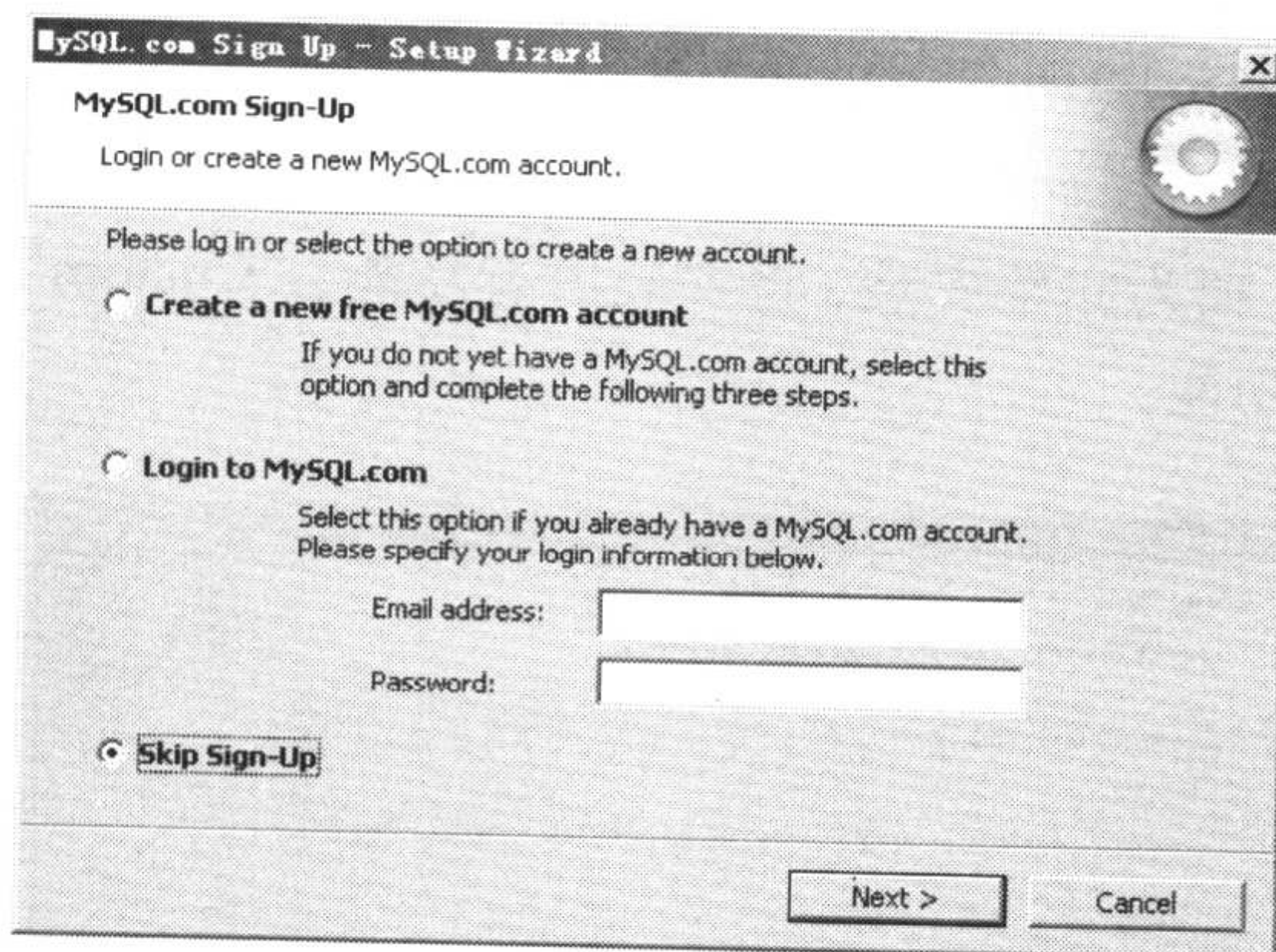


图 16-18 注册对话框

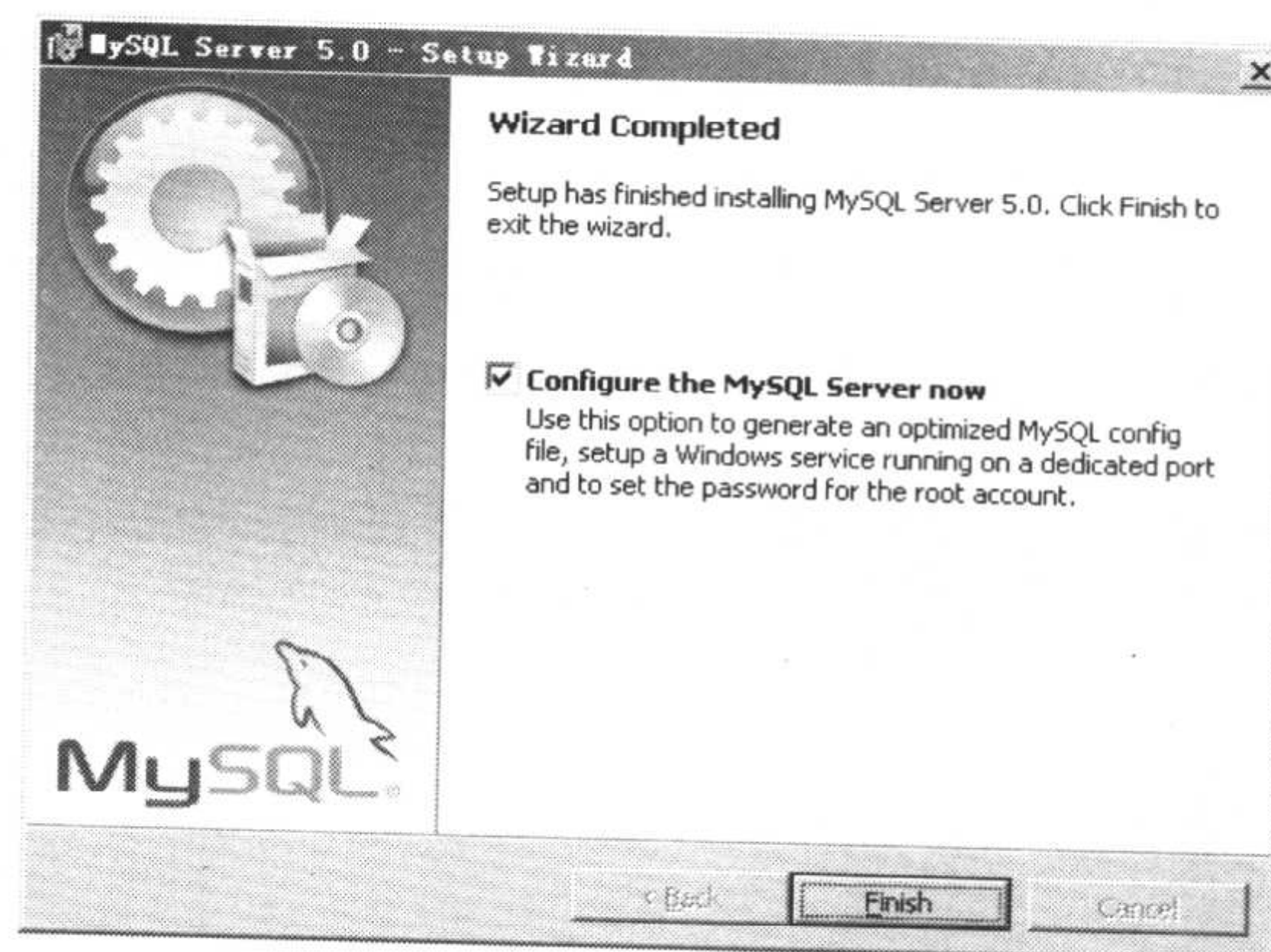


图 16-19 完成安装

(6) 单击【Finish】按钮完成安装，进入数据库设置界面，如图 16-20 所示。

(7) 单击【Next】按钮，进入设置类型界面，由于是初次安装，因此选中【Standard Configuration】单选框，如图 16-21 所示。

(8) 单击【Next】按钮，选中【Include Bin Directory in Windows PATH】复选框，将 MySQL 添加到 PATH 环境变量中，如图 16-22 所示。

(9) 在密码框中填写登录到 MySQL 的密码，该密码将在连接数据库时使用。选中【Create An Anonymous Account】复选框，创建匿名用户。

第16章 Python 与数据库

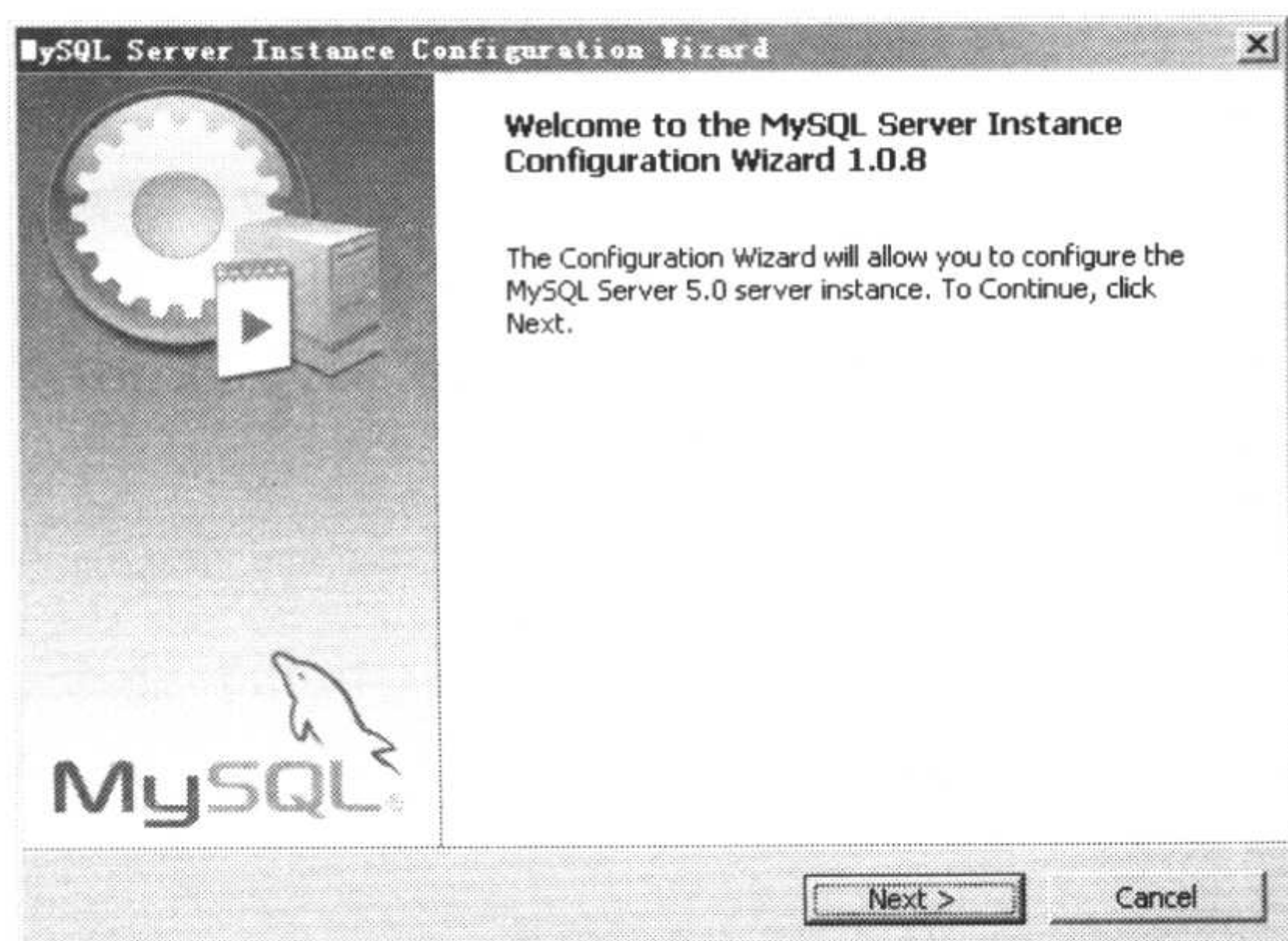


图 16-20 数据库设置

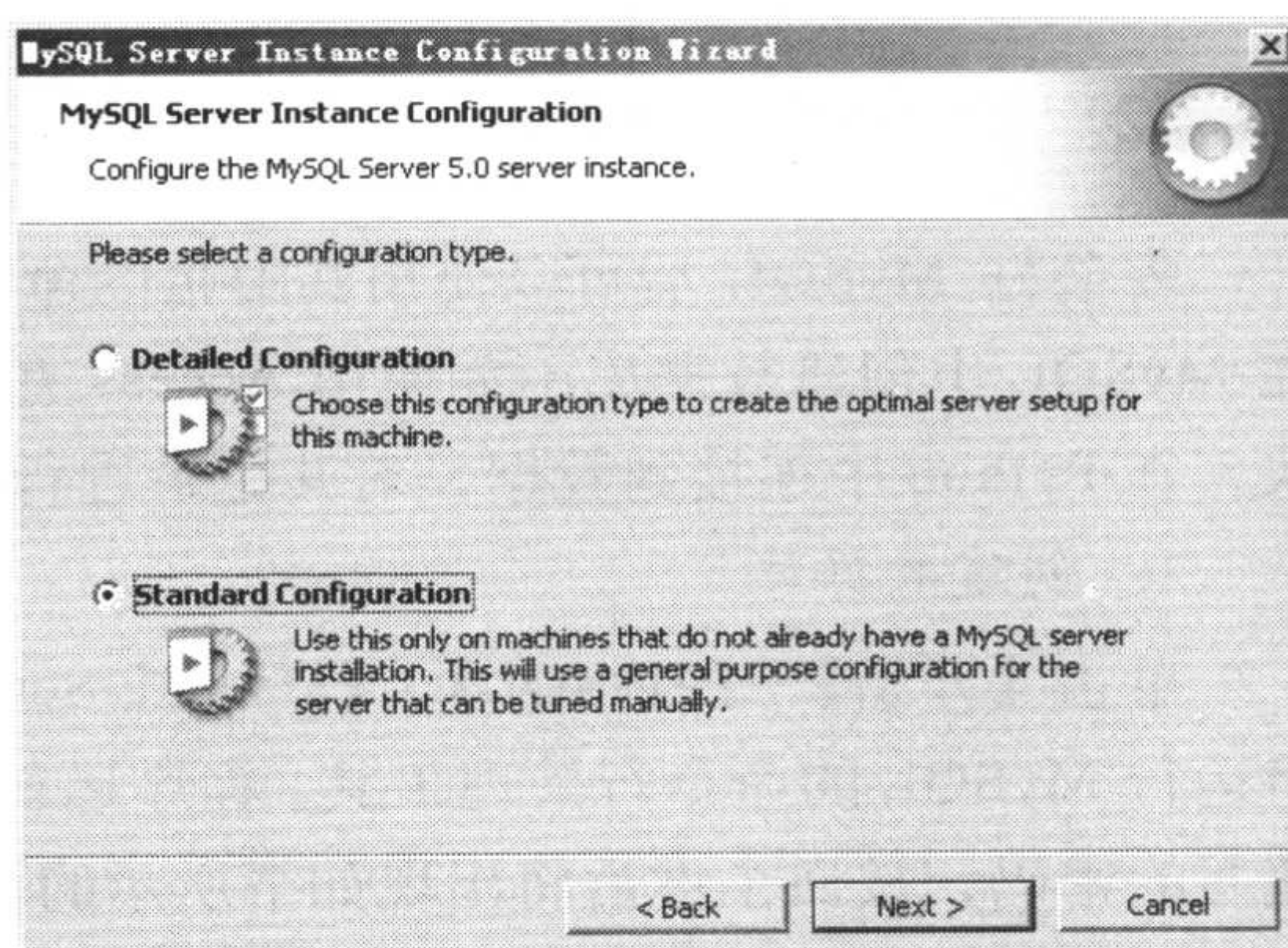


图 16-21 设置类型

(10) 单击【Next】按钮进入下一步安装，如图 16-23 所示。单击【Execute】按钮，完成安装，如图 16-24 所示。安装完之后的界面如图 16-25 所示。

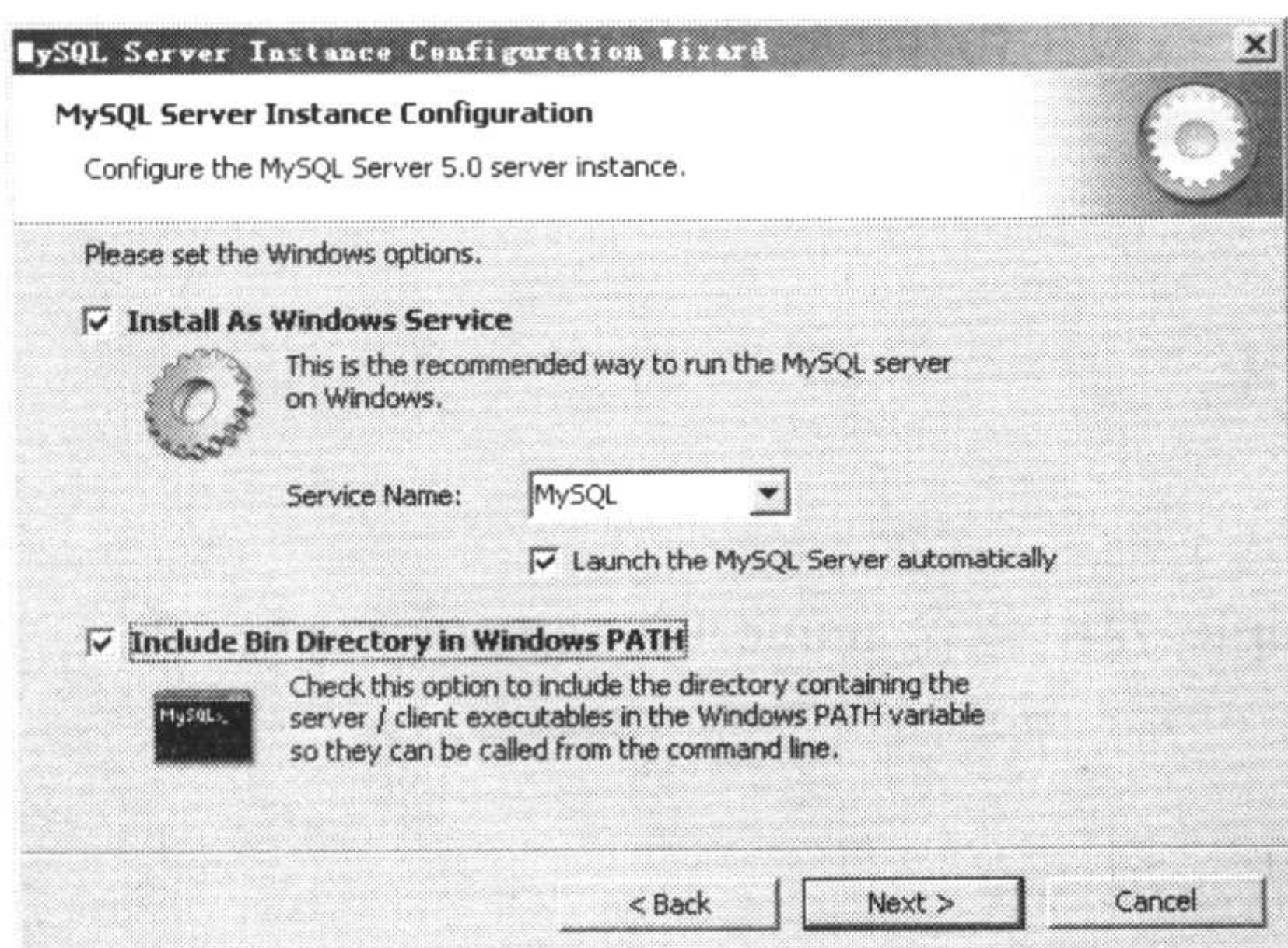


图 16-22 设置选项

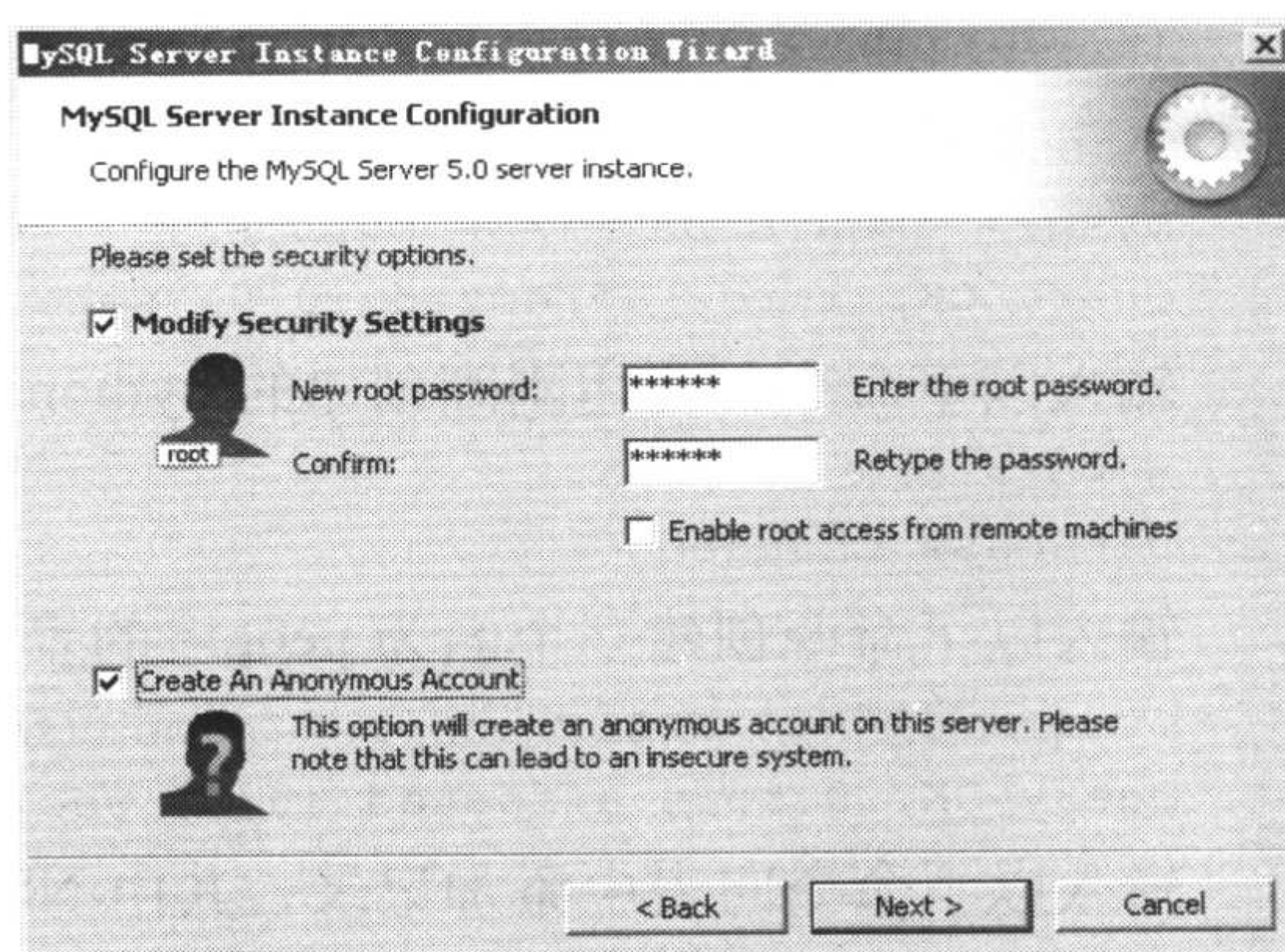


图 16-23 设置密码

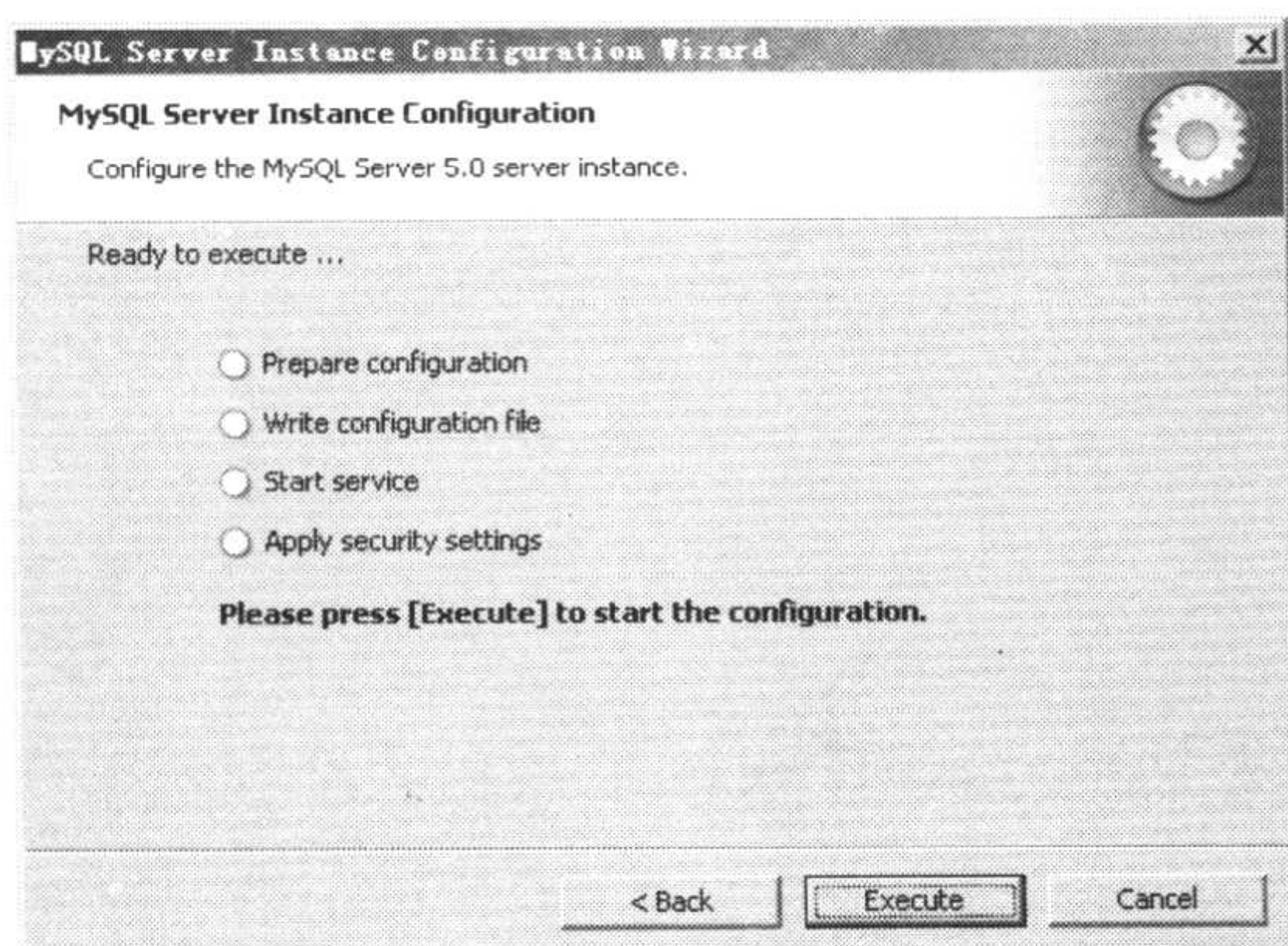


图 16-24 设置生效

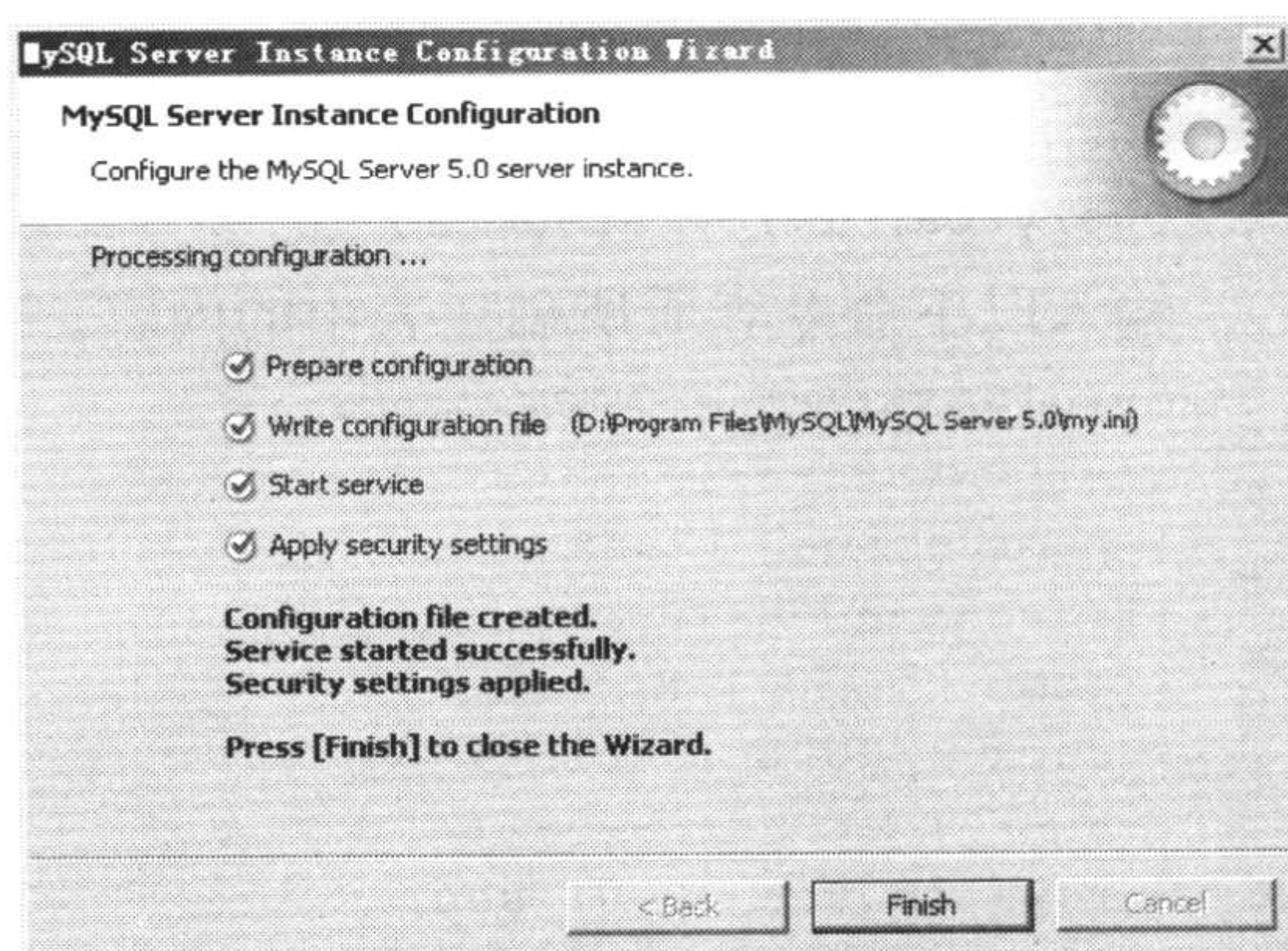


图 16-25 完成设置

16.2.2 连接到 MySQL

安装好 MySQL 后可以使用其附带的命令行工具创建数据库。由于是命令行工具，因此在 MySQL 中创建数据库不如使用 Access 直观。创建好数据库后，就可以使用 MySQLdb 模块，在 Python 中连接到数据，对其记录进行操作了。

1. 创建数据库

单击【开始】|【MySQL】|【MySQL Server 5.0】|【MySQL Command Line Client】命令，将运行 MySQL 的命令行管理工具。程序运行后会提示输入密码，此时输入安装 MySQL 时所填写的密码，按回车键后将出现如下所示的提示。

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

在“mysql>”提示符下输入以下命令创建一个名为“python”的数据库（斜体部分为用户输入部分）。

```
mysql> CREATE DATABASE python;
Query OK, 1 row affected (0.01 sec)
```

输入以下命令，使用刚刚创建的 python 数据库。

```
mysql> USE python;
Database changed
```

输入以下命令创建一个名为 people 的表，people 表中包含 name 项、age 项和 sex 项。

```
mysql> CREATE TABLE people (name VARCHAR(30), age INT, sex CHAR(1));
Query OK, 0 rows affected (0.20 sec)
```

输入以下命令向表中添加记录，其中 NULL 表示项为空。

```
mysql> INSERT INTO people VALUES('Tom',20,'M');
Query OK, 1 row affected (0.06 sec)
```

```
mysql> INSERT INTO people VALUES('Jack',NULL,NULL);
Query OK, 1 row affected (0.06 sec)
```

输入以下命令查看所创建表中的内容。

```
mysql> SELECT * FROM people;
+-----+-----+-----+
| name | age | sex |
+-----+-----+-----+
| Tom  | 20  | M   |
| Jack | NULL | NULL |
+-----+-----+-----+
2 rows in set (0.01 sec)
```

完成数据库创建后，可以使用“exit”命令退出命令行。

2. 安装 MySQLdb

在 Python 中使用 MySQL 的数据库需要安装 MySQLdb 模块。从其官方网站 <http://sourceforge.net/projects/mysql-python> 下载 Windows 下的安装程序。以 Python 2.5 为例安装步骤如下所示。

(1) 下载 MySQL-python-1.2.2.win32-py2.5.exe 文件，下载完成后双击该文件，如图 16-26 所示。

(2) 单击【下一步】按钮，如图 16-27 所示，安装程序将自动搜索安装的 Python。

(3) 单击【下一步】按钮，即可完成安装。

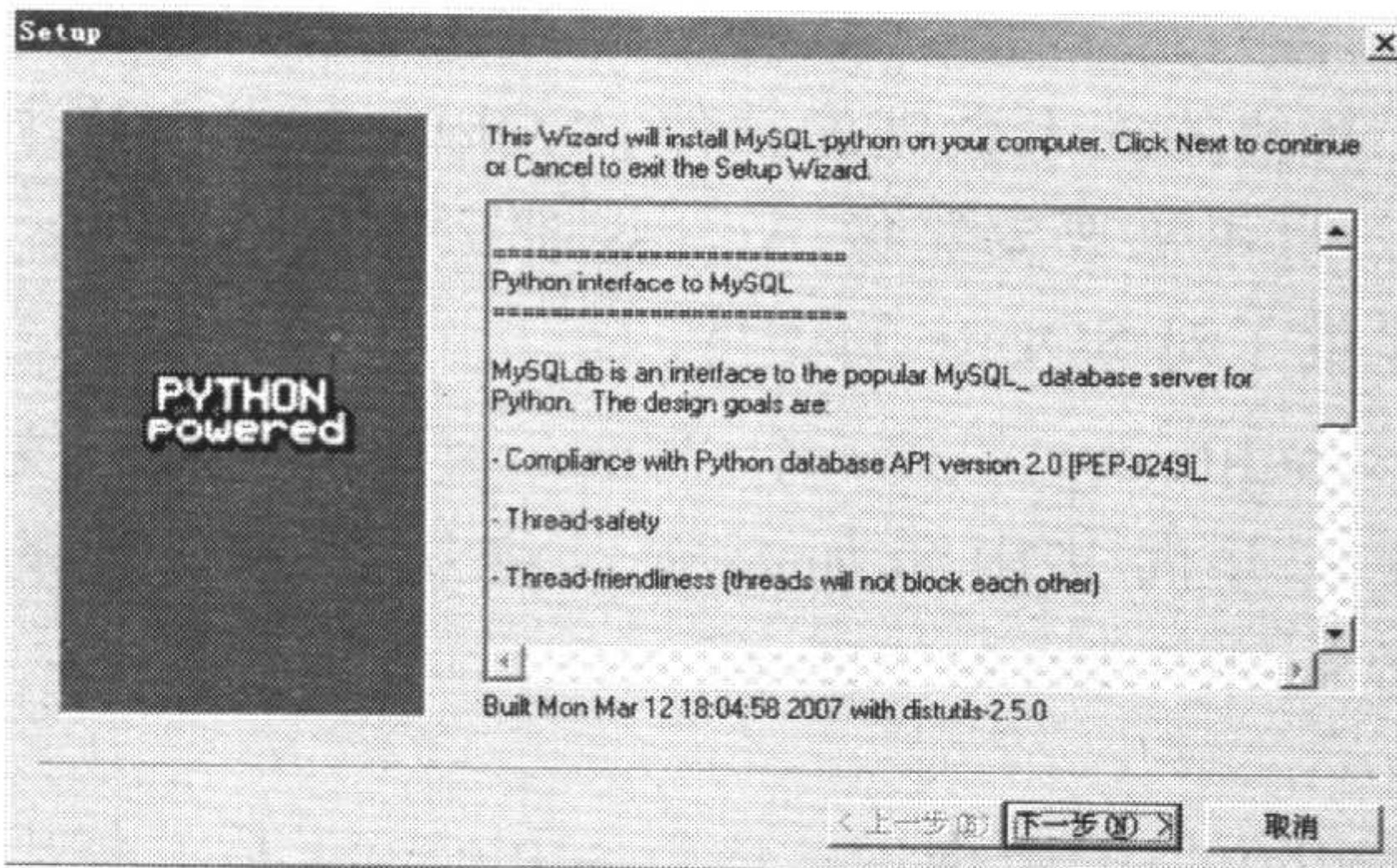


图 16-26 安装 MySQLdb

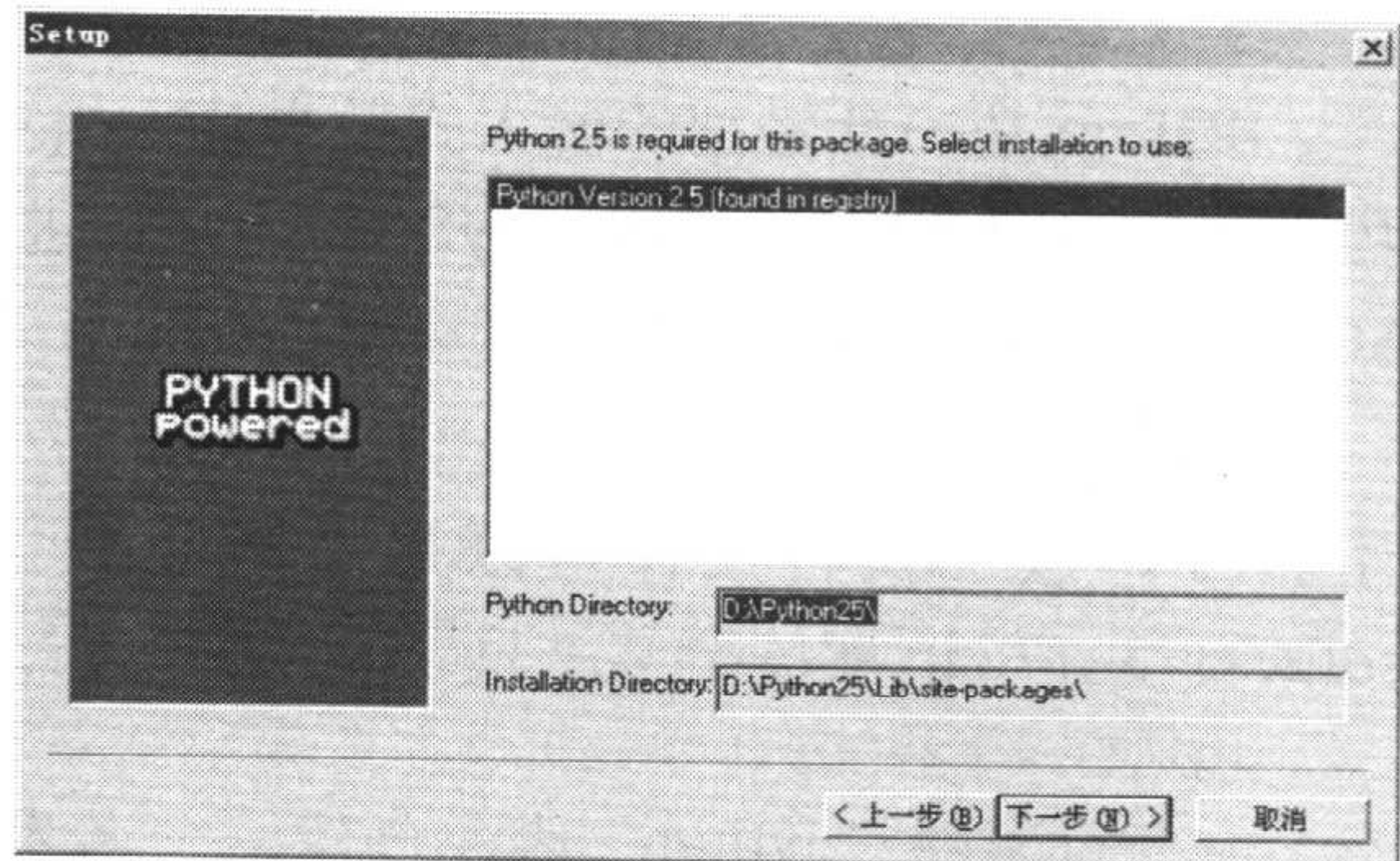


图 16-27 安装路径

当完成安装后可以在交互式 Python Shell 下输入如下所示的语句。

```
import MySQLdb
```

如果上述语句成功运行，则说明 MySQLdb 已经安装成功。

3. 在 Python 中使用 MySQL 数据库

使用 MySQLdb 连接到 MySQL 数据库和使用 ODBC 连接到 Access 数据库类似。首先使用 MySQLdb 模块的 connect 方法连接到 MySQL 守护进程。connect 方法将返回一个数据库连接。使用数据库连接的 cursor 方法可以获得当前数据库的游标。然后就可以使用游标的 Execute 方法执行 SQL 语句，完成对数据库的操作。同样，当完成操作应调用 close 方法关闭游标和数据库连接。如下所示的 PyMySQL.py 脚本使用 MySQLdb 连接到所创建的 python 数据库。

```
# -*- coding:utf-8 -*-
# file: PyMySQL.py
#
```

```
import MySQLdb
db = MySQLdb.connect(host='localhost',
                     user='root',
                     passwd='python',
                     db='python')
```

```
# 导入 MySQLdb 模块
# 连接到数据库，服务器为本机
# 用户名为 root
# 密码为 python
# 数据库名为 python
```



```

cur = db.cursor()                                # 获得数据库游标
cur.execute('insert into people (name,age,sex) values (\Jee\',21,\F\')')
                                                    # 执行 SQL 语句, 添加记录
r = cur.execute('delete from people where age=20')
                                                    # 执行 SQL 语句, 删除记录
con.commit()                                     # 提交事务
r = cur.execute('select * from people')          # 执行 SQL 语句, 获取记录
r = cur.fetchall()                              # 获取数据
print r                                          # 输出数据
cur.close()                                     # 关闭游标
db.close()                                      # 关闭数据库连接

```

16.3 嵌入式数据库 SQLite

SQLite 是一款轻型的嵌入式数据库, 相对于其他的庞大数据库软件, SQLite 显得十分小巧。使用 SQLite 不需要像 MySQL 的守护进程, 也不需要安装像 Access 那么庞大的软件。SQLite 可以满足一般的数据库应用, Python 也提供了对 SQLite 的支持。

SQLite 不需要安装, 从其官方网站 <http://www.sqlite.org> 下载一个 Windows 版的可执行文件即可。该可执行文件可以用于创建数据库, 并向其中添加内容。将从官方下载的 `sqlite-3_3_17.zip` 解压至某一目录。从命令行运行 `sqlite3.exe` 如下所示。

```
sqlite3.exe python
```

命令行参数 “python” 表示创建一个名为 “python” 的数据库。运行 `sqlite3.exe` 后, 将出现如下所示的提示。

```

SQLite version 3.3.17
Enter ".help" for instructions
sqlite>

```

在 “sqlite>” 命令提示符后输入以下命令可以在 “python” 中创建一个名为 “people” 的表。

```
sqlite> CREATE TABLE peple (name VARCHAR(30), age INT, sex CHAR(1));
```

输入如下所示的命令向 people 表中插入内容。

```

sqlite> INSERT INTO people VALUES ('Tom', 20, 'M');
sqlite> INSERT INTO people VALUES ('Jack', 21, 'M');

```

输入如下所示的命令查看 people 表中的内容。

```

sqlite> SELECT * FROM people;
Tom|20|M
Jack|21|M

```

完成对数据库的操作后可以使用 “.exit” 命令退出命令行, 此时就可以在 Python 中使用所创建的 “python” 数据库了。在 Python 中使用 SQLite 数据库和使用 ODBC 及 MySQL 数据库的过程类似。首先导入 `sqlite3` 模块, 由于 SQLite 不需要服务器, 因此直接使用 `connect` 方法打开数据库即可。`connect` 方法返回一个数据库连接对象, 使用其 `cursor` 方法可以获得一个游标, 然后就可以对记录进行操作。在完成操作后, 应使用 `close` 方法关闭游标和数据库连接。如下所示的 `PySqlite.py` 脚本在 Python 中使用 SQLite 数据库。

```
# -*- coding:utf-8 -*-
# file: PySqlite.py
#
import sqlite3
con = sqlite3.connect('python')
cur = con.cursor()
cur.execute('insert into people (name,age,sex) values (\''Jee\'',21,\''F\'')')
r = cur.execute('delete from people where age=20')
con.commit()
cur.execute('select * from people')
s = cur.fetchall()
print s
cur.close()
con.close()
```

导入 sqlite3 模块
连接到数据库
获得数据库游标
执行 SQL 语句, 添加记录
执行 SQL 语句, 删除记录
提交事务
执行 SQL 语句, 获取记录
获得数据
输出数据
关闭游标
关闭数据库连接

第 17 章 Python Web 应用

Python 可以和 ASP、PHP 等一样应用于 Web 服务，还可以像 VBScript、JavaScript 一样作为脚本嵌入到 ASP 中。Windows IIS 支持“.py”文件，可以使用“.py”文件代替“.asp”文件。除了在 Windows IIS 中使用 Python 以外，在 Apache 中也可以使用 Python。另外，还可以使用 Python 编写的 Web 服务器 Zope。目前，有很多基于 Python 的 Web 框架技术，例如，Plone、Django、TurboGears 等，使用 Web 框架可以简化 Web 应用程序设计。

17.1 开源 Web 应用服务器 Zope

Zope 是一个开源 Web 应用服务器，其主要使用 Python 编写。Zope 提供了完善的功能，可以实现 ASP、PHP、JSP 的功能，构建各种类型的 Web 应用。在 Zope 里可以使用数据库、模板等，Zope 使用了面向对象的思想，在 Zope 里一切都被称为对象。

17.1.1 安装 Zope

Zope 可以运行在多个操作系统下，在 Windows 下安装 Zope 可以到其官方网站 <http://www.zope.org> 下载 Windows 版的安装程序。Zope 官方网站提供了附带 Python 的 Zope 安装程序，使得 Zope 的安装较为简单。以 Zope-2.10.3-win32.exe 为例安装过程如下。

- (1) 双击运行 Zope-2.10.3-win32.exe，如图 17-1 所示。
- (2) 单击【Next】按钮进入安装路径选择界面，此处可以根据需要重新设置 Zope 的安装路径，如图 17-2 所示。
- (3) 单击【Next】按钮进入安装内容选择界面，此处保持默认设置，如图 17-3 所示。
- (4) 单击【Next】按钮进入设置服务界面，此处保持默认设置，将 Zope 设置为 Windows 的服务，自动启动，如图 17-4 所示。如果未将 Zope 设置为 Windows 的服务，则每次使用 Zope 时都需要手工启动。
- (5) 单击【Next】按钮进入服务安装路径选择界面，此处可以根据需要修改，如图 17-5 所示。
- (6) 单击【Next】按钮进入密码设置界面，该密码为登录 Zope 的管理密码，如图 17-6 所示。

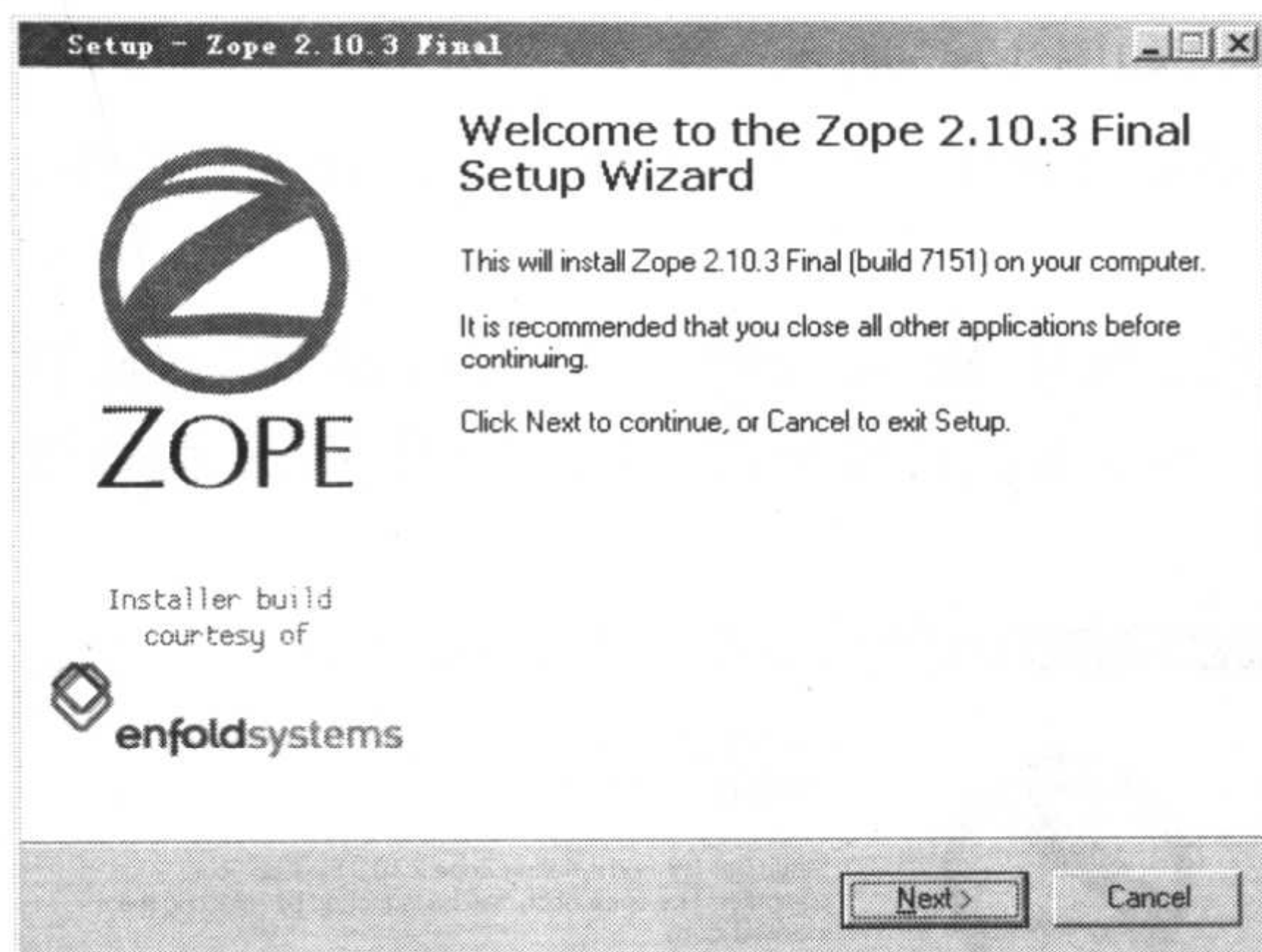


图 17-1 安装 Zope

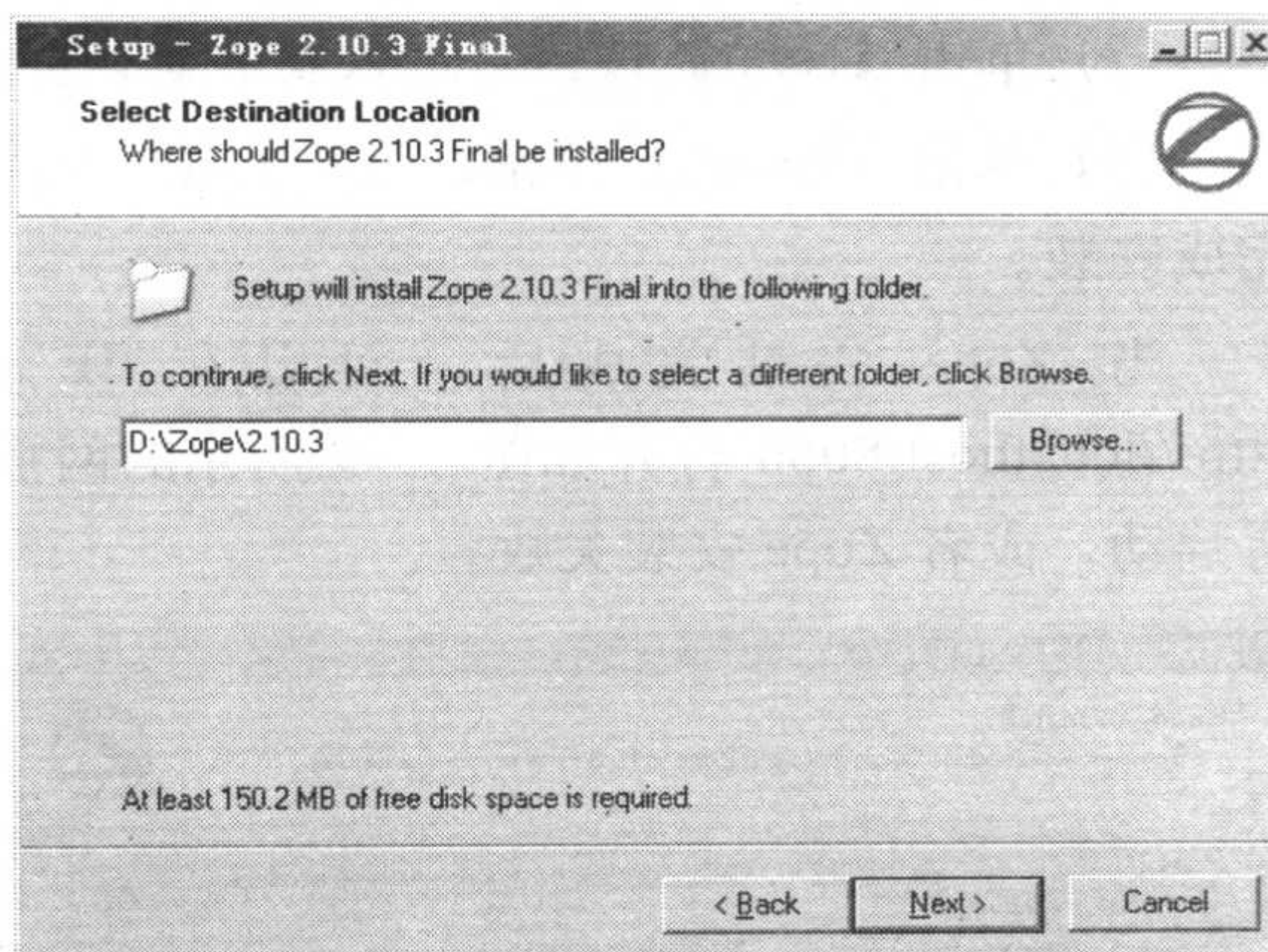


图 17-2 选择安装路径

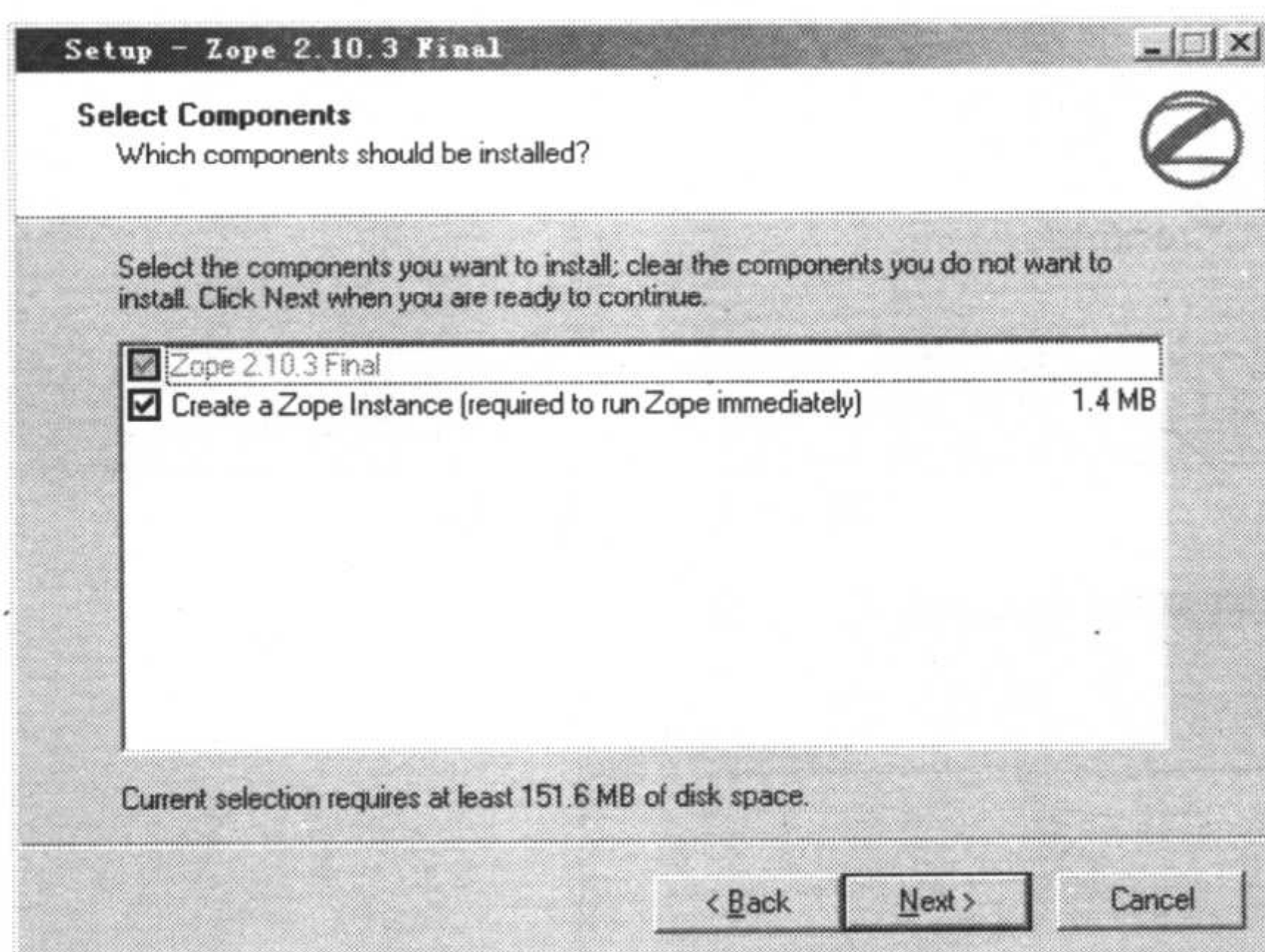


图 17-3 选择安装内容

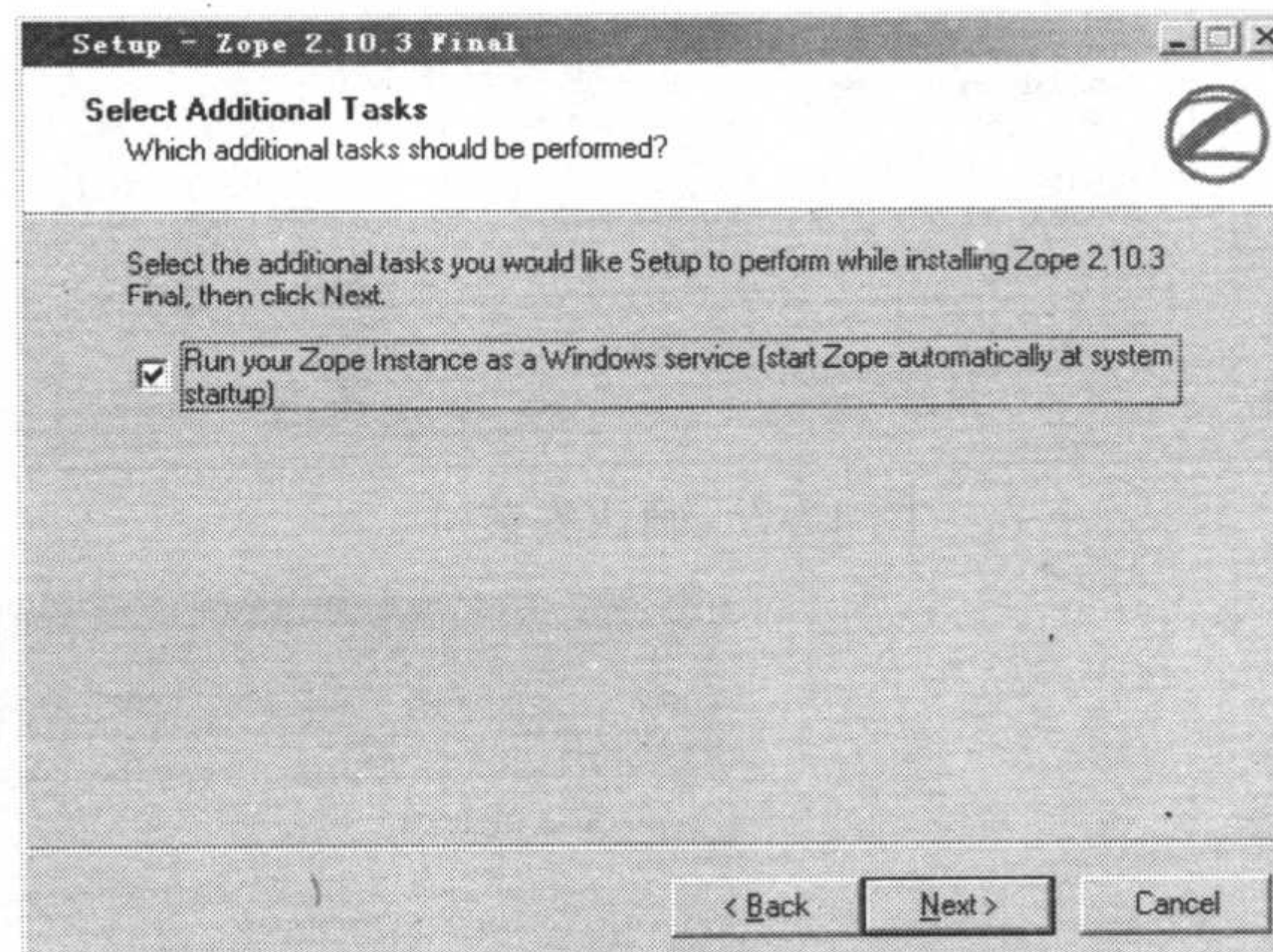


图 17-4 设置服务

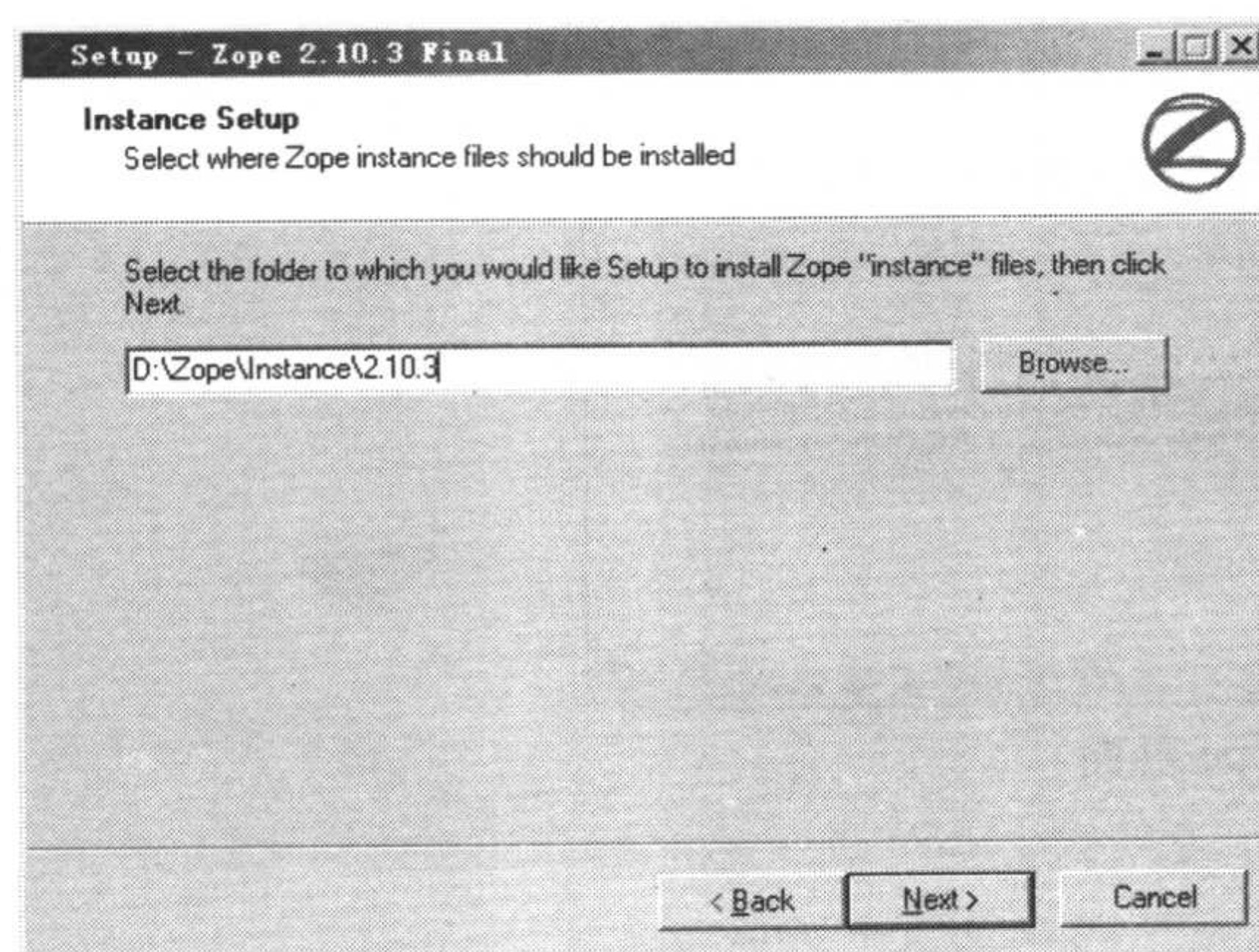


图 17-5 服务安装路径

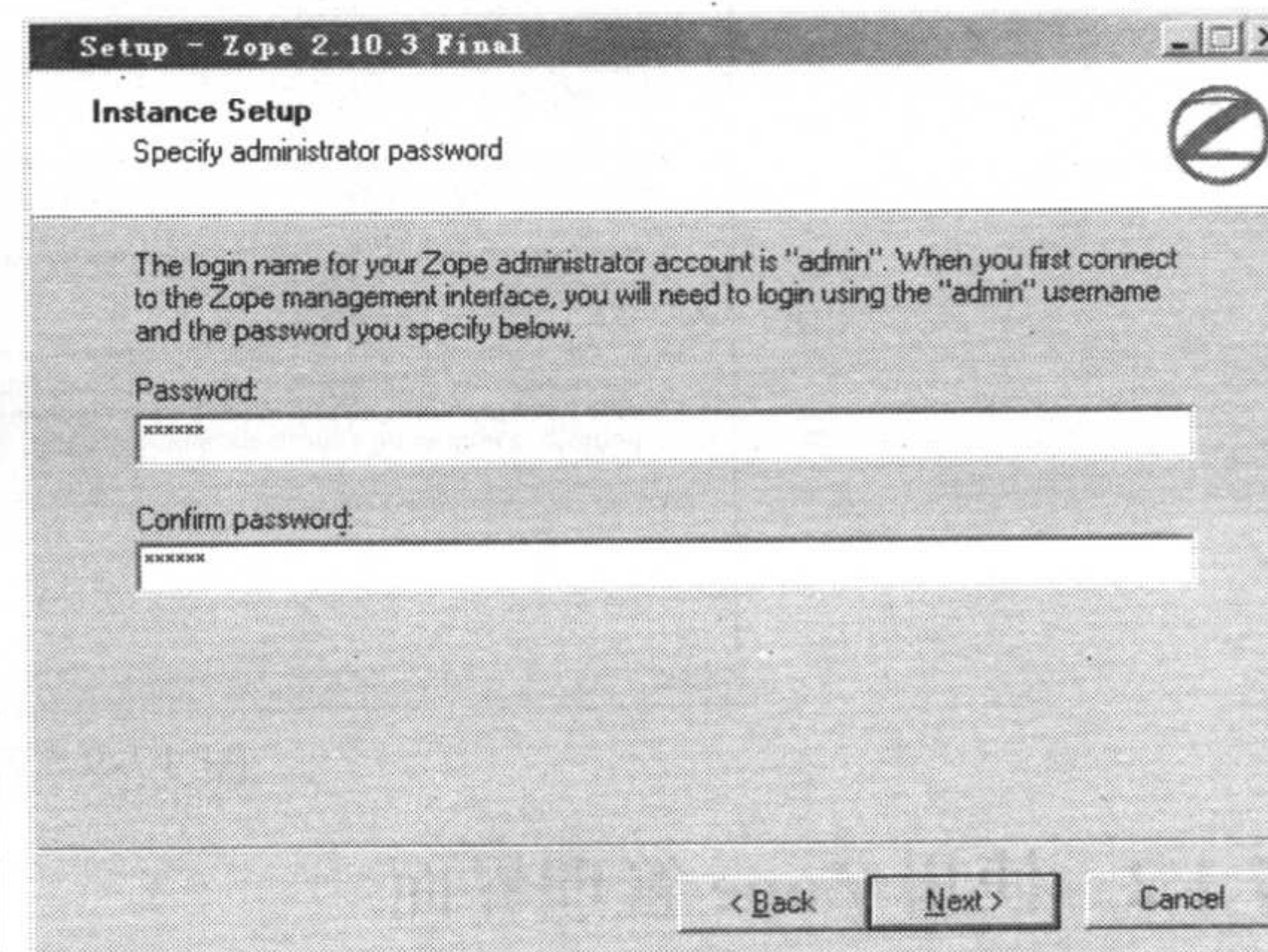


图 17-6 设置管理密码

(7) 单击【Next】按钮，进入安装确认界面，如图 17-7 所示。

(8) 单击【Install】按钮，进行安装，安装完成后如图 17-8 所示，单击【Finish】按钮，完成安装。

当 Zope 安装完成后，可以在 IE 地址栏中输入 `http://127.0.0.1:8080`，或者 `http://localhost:8080` 打开如图 17-9 所示的网页。如果未能打开页面，则可能是 Zope 服务没有启动，或者 Zope 安装失败。

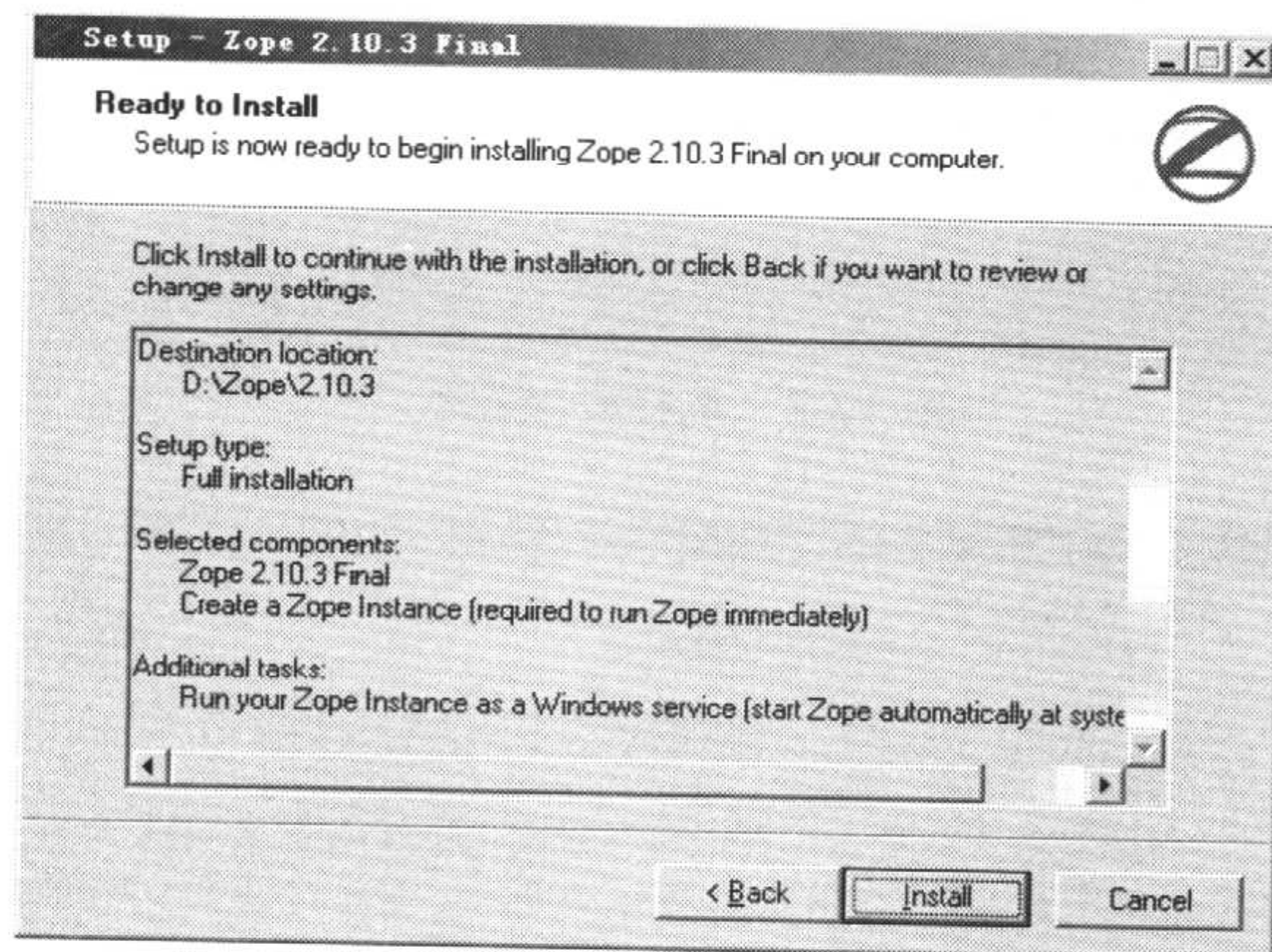


图 17-7 确认安装

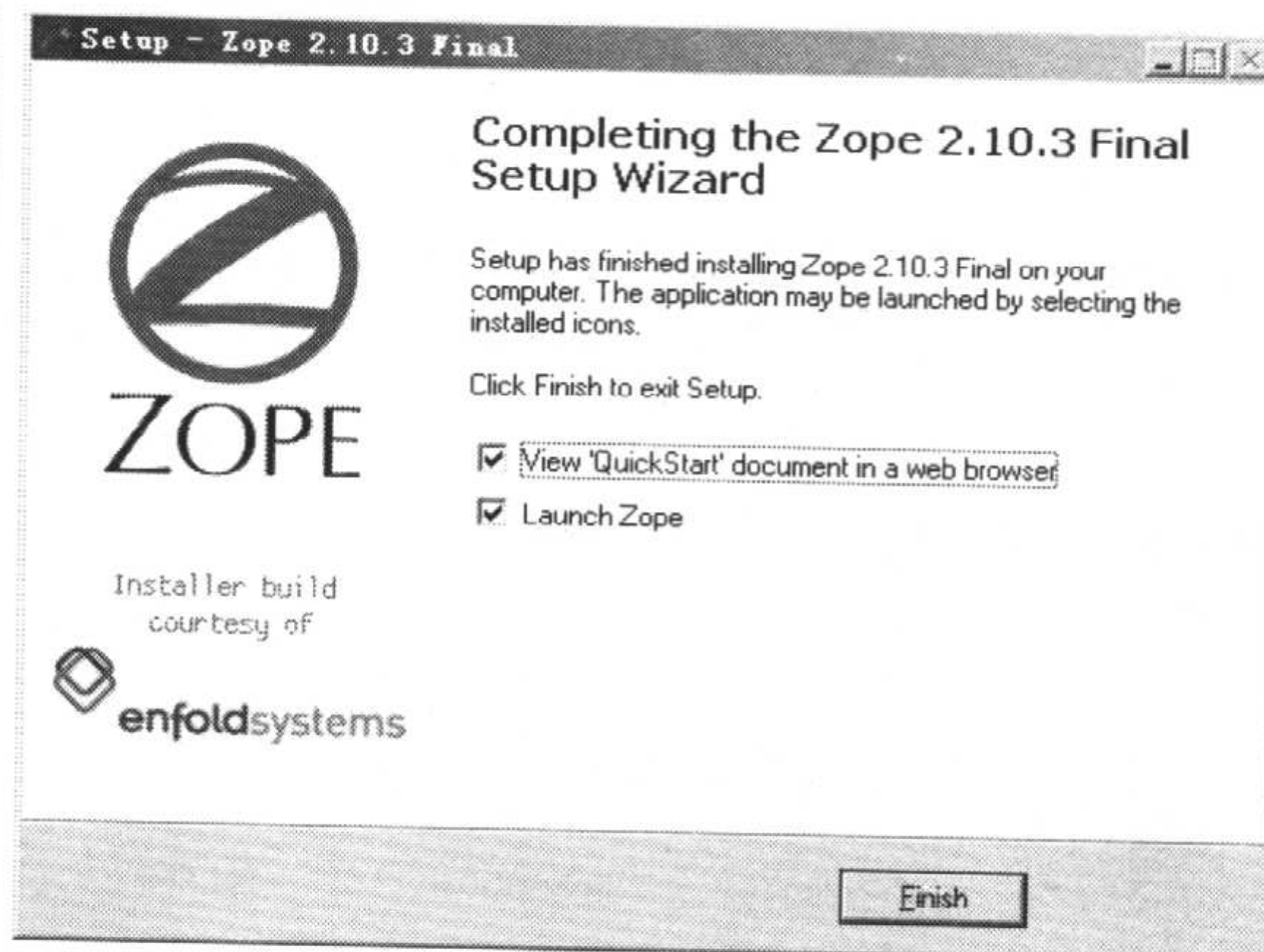


图 17-8 完成安装

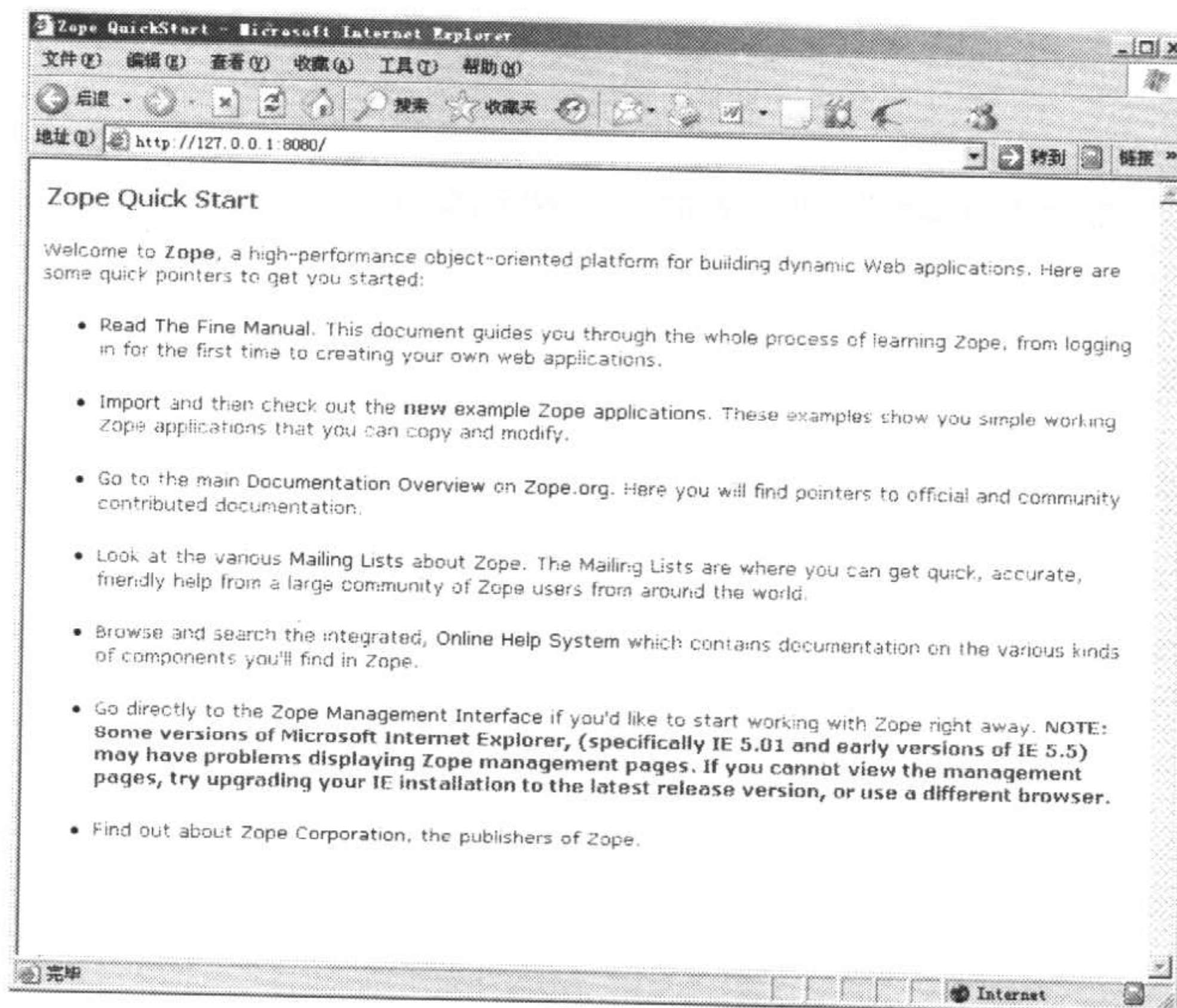


图 17-9 Zope 页面

17.1.2 使用 Zope 管理界面

安装好 Zope 后，在 IE 地址栏中输入 `http://127.0.0.1:8080/manage` 可以打开 Zope 的管理

第17章 Python Web 应用

界面。Zope 将要求输入密码，即安装 Zope 时所创建的密码，用户名为“admin”，如图 17-10 所示。单击【确定】按钮后，将进入 Zope 管理界面，如图 17-11 所示。Zope 管理界面由以下几个部分组成。

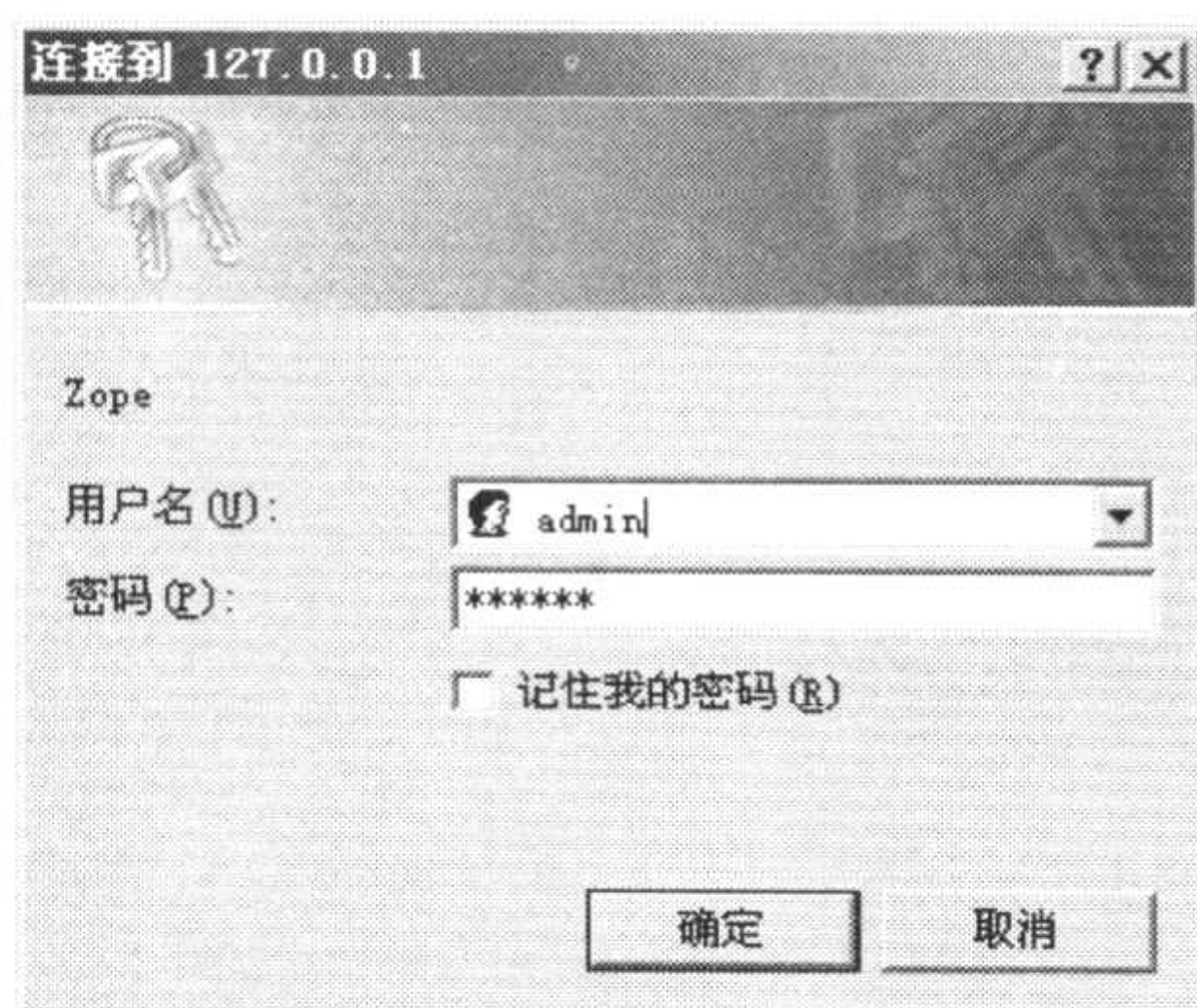


图 17-10 登录 Zope

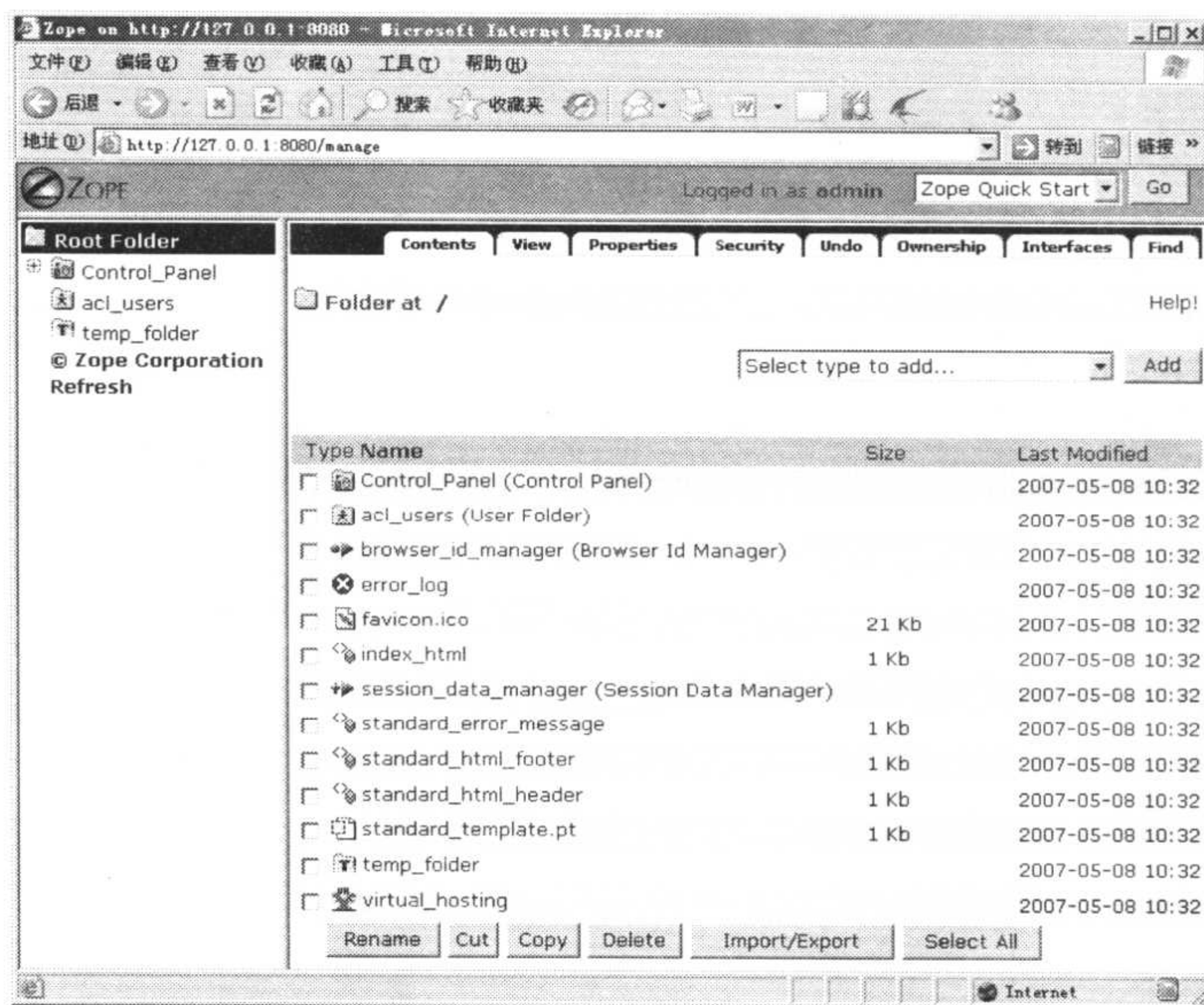


图 17-11 Zope 管理界面

- 对象管理。
- 控制面板。
- 用户管理。
- 错误日志。

其中，对象管理主要是添加、删除、修改对象，修改对象属性、权限等。对象管理是 Zope 的最主要功能，通过对象管理可以完成站点的建设。在控制面板中可以查看服务器的状态，关闭或者重新启动服务器。在控制面板中还可以设置数据库。用户管理可以用于创建新的 Zope 管理用户。错误日志记录了 Zope 所出现的错误。在 Zope 管理界面中添加页面的步骤如下。

- (1) 选择【Select type to add】下拉列表框中的【DTML Document】，如图 17-12 所示。
- (2) 单击【Add】按钮，在【Id】文本框中填写“python”，在【Title】文本框中填写“Python”，如图 17-13 所示。
- (3) 单击【Add and Edit】按钮，将多行文本框中原有内容删除，输入如下所示内容，如图 17-14 所示。

```
<head>
```



```
<title>
Python
</title>
</head>
<body>
<h1>Python and Zope</h1>
<p>
<a href="http://www.python.org">Python Home Page</a>
</body>
</html>
```

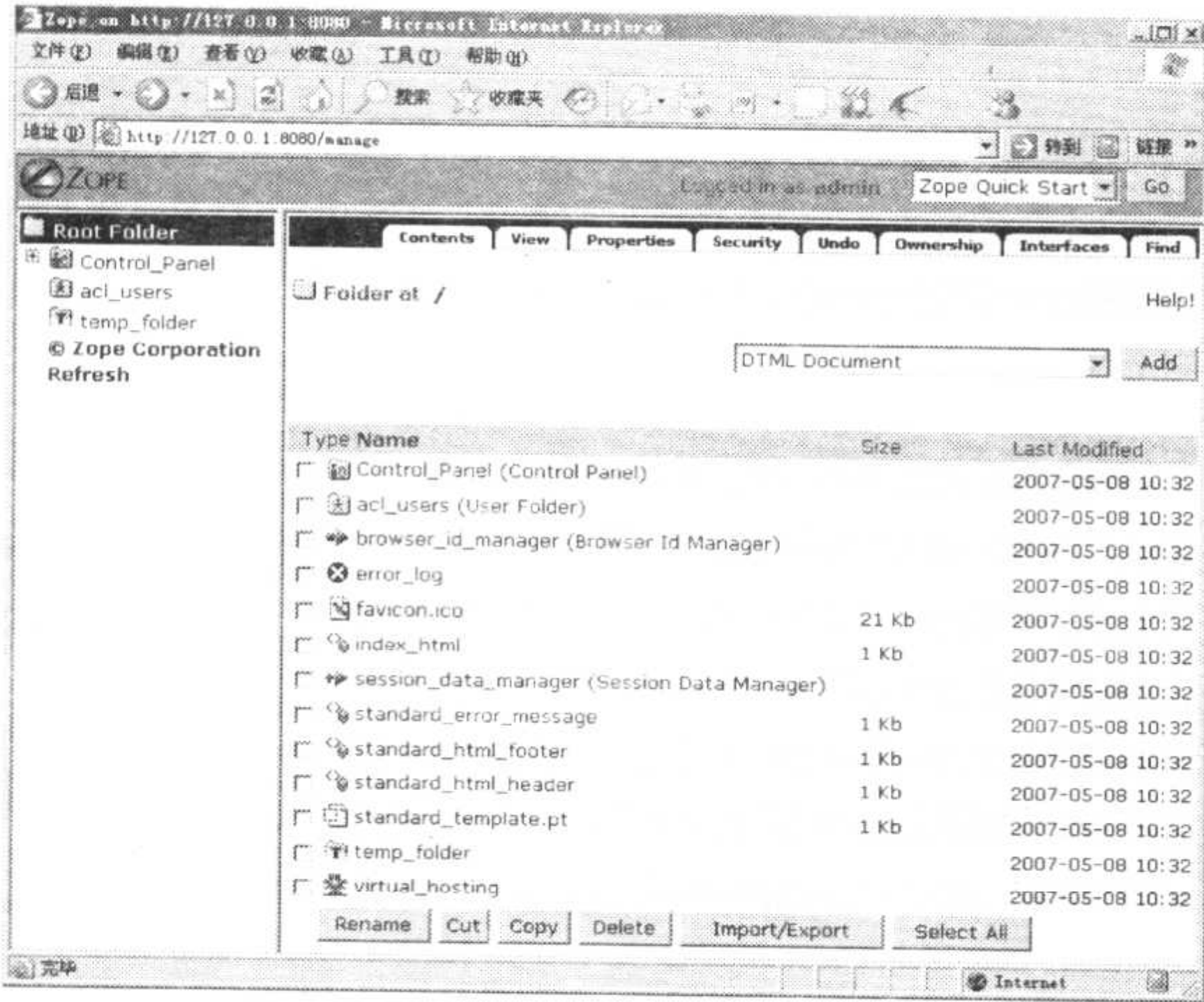


图 17-12 选择类型

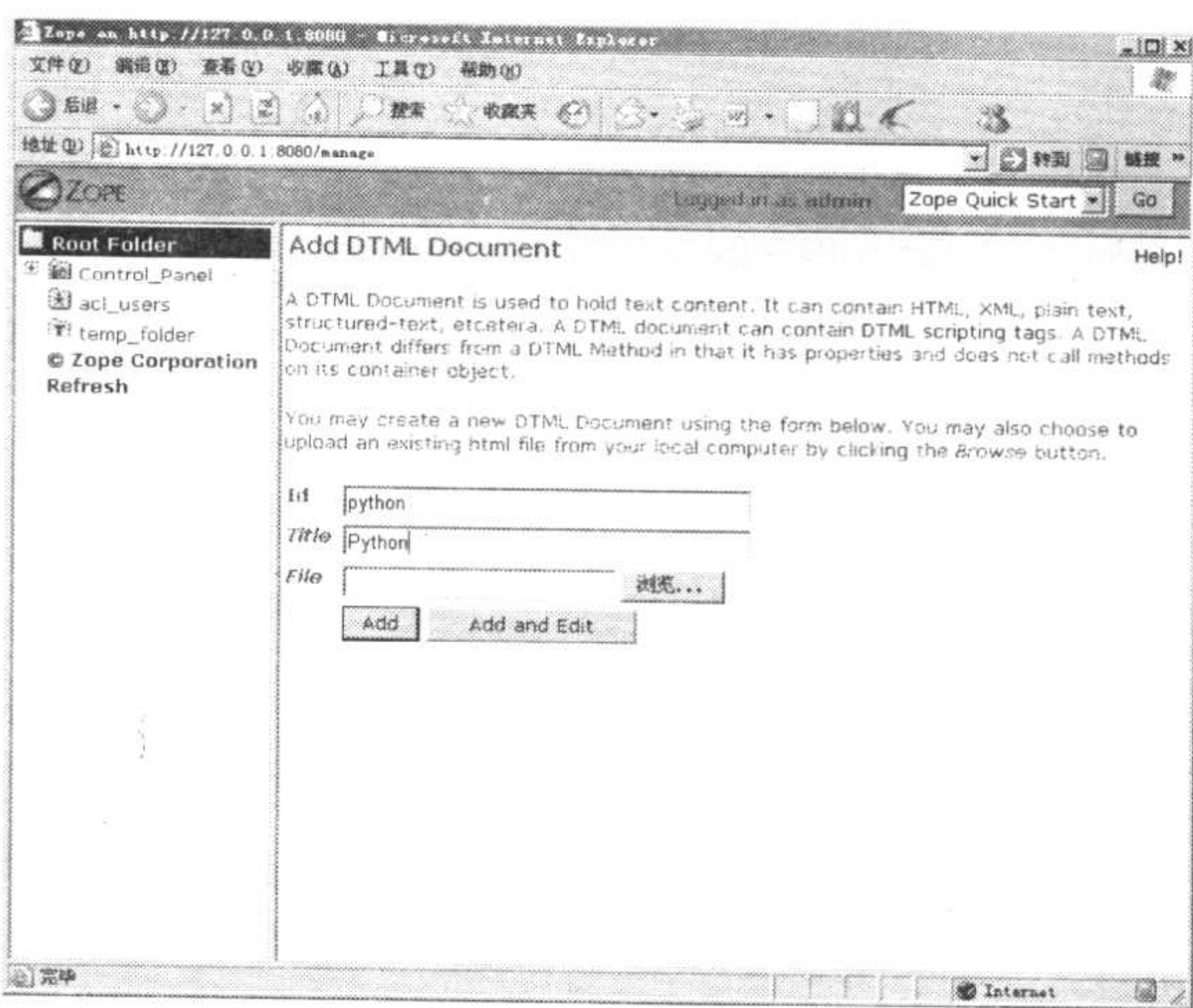


图 17-13 输入文件 Id

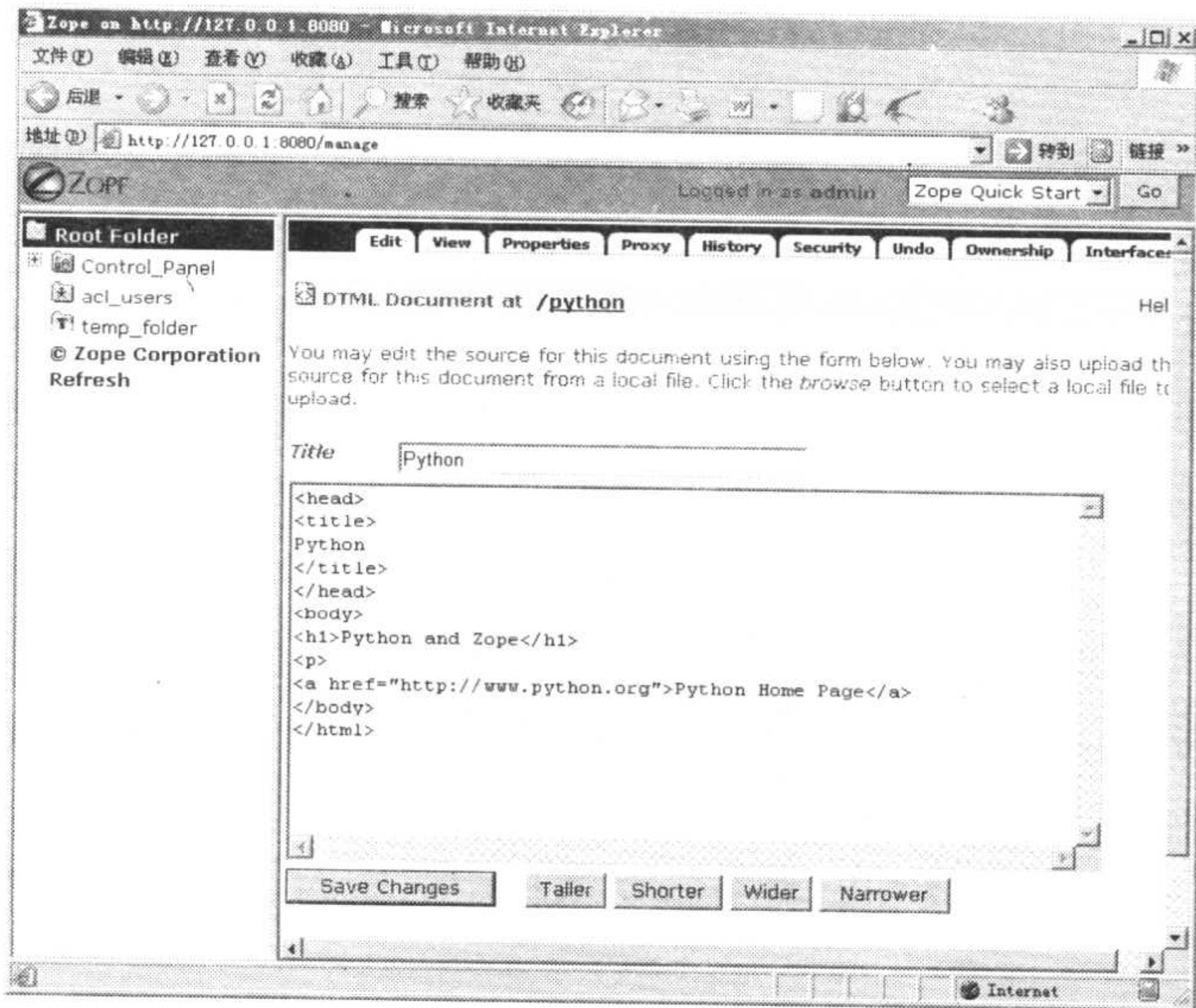


图 17-14 输入文件内容

(4) 单击【Save Changes】按钮，保存内容。此时可以通过填写的“Id”访问刚才创建的文件。在 IE 地址栏中输入 `http://127.0.0.1:8080/python`，如图 17-15 所示。

在 Zope 中使用中文需要对 Zope 进行设置。假设 Zope 服务安装在“D:\Zope\Instance\2.10.3”目录，打开“D:\Zope\Instance\2.10.3\etc”目录中的“zope.conf”文件，将“default-zpublisher-encoding utf-8”前的注释去掉。如果没有“default-zpublisher-encoding utf-8”，则可以向“zope.conf”文件中添加，包含中文的文件应该保存成“Utf-8”的格式，然后在图 17-13 中单击【File】右侧的【浏览】按钮上传到 Zope 中。

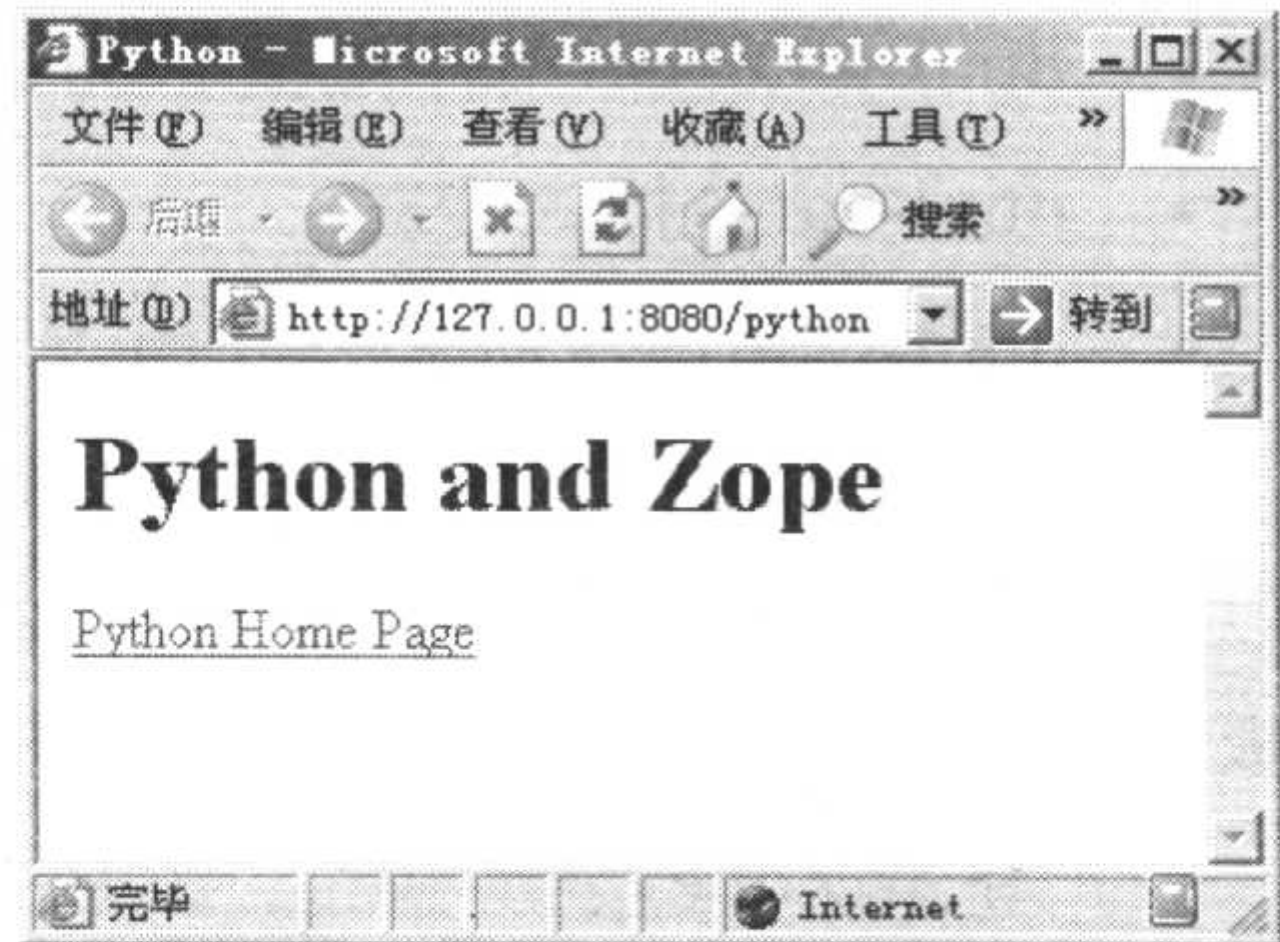


图 17-15 浏览所添加的文件

17.1.3 创建模板

在 Zope 里创建模板可以使用 DTML (Document Template Markup Language) 和 ZPT (Zope Page Template)。DTML 是 Zope 中较早的模板语言，适合邮件模板、CSS 等的编写。ZPT 提供了创建页面模板更为有效的方式。

1. 使用 DTML

DTML 是由 DTML 命令列组成的，DTML 命令列是以“dtml-”为开头的标签。DTML 命令列可以和 HTML 混合在一起使用。使用 DTML 可以重复使用内容，保证内容格式统一，充分利用资源。使用 DTML 的标签类似于 JavaScript、VBScript 可以嵌入到 HTML 中实现编程。不同的是，DTML 是在服务器端执行。常用的 DTML 标签有如下几种。

- dtml-call: 调用方法。
- dtml-comment: 注释 DTML，为语句块。
- dtml-if: 条件测试，为语句块，包含 dtml-elif 和 dtml-else。
- dtml-in: 循环语句，为语句块。
- dtml-let: 定义 DTML 变量，为语句块。
- dtml-mime: 创建 MIME 编码，为语句块，用于处理邮件格式。
- dtml-raise: 引发异常，为语句块。
- dtml-return: 返回数据。
- dtml-sendmail: 发送邮件，为语句块。
- dtml-sqlgroup: 处理 SQL 语句，为语句块。
- dtml-sqltest: 测试 SQL 代码中的变量值。
- dtml-sqlvar: 向 SQL 代码中插入变量。

- dtml-tree: 创建树组件, 为语句块。
- dtml-try: 捕捉异常, 为语句块, 包含 dtml-except 和 dtml-finally。
- dtml-unless: 条件测试, 为语句块。
- dtml-var: 插入变量。
- dtml-with: 在命名空间中查找变量, 为语句块。

除了上述的标签以外, DTML 还提供了内置函数用于完成简单的任务。DTML 的标签可以和 HTML 的标签混合使用。如下所示的步骤是用 DTML 在 Zope 中创建一个简单的模板。

(1) 登录 Zope 管理界面, 添加一个 Id 为 “head” 的 DTML Document 文件, 其内容如下所示。

```
<table border="1">
  <tr>
    <td><font size="10">Python Web</font></td>
  </tr>
</table>
```

(2) 添加一个 Id 为 “foot” 的 DTML Document 文件, 其内容如下所示。

```
<table border="1">
  <tr>
    <td align="right">Powered by Zope</td>
  </tr>
</table>
```

(3) 添加一个 Id 为 “dpt” 的 DTML Document 文件, 其内容如下所示。

```
<dtml-var head>
<h2><dtml-var title_or_id></h2>
<p>
This is the <dtml-var id> Document.
</p>
<dtml-var foot>
```

在 “dpt” 中主要使用 dtml-var 标签, 在其头部插入所创建的 “head” 文件, 在其结尾插入所创建的 “foot” 文件。保存文件后, 在 IE 地址栏中输入 <http://127.0.0.1:8080/dpt>, 如图 17-16 所示。

2. 使用 ZPT

在 ZPT 中使用的 TAL (Template Attribute Language) 是由以 “tal:” 开头的一系列标记、属性以及其他相关的值组成的。TAL 可以嵌套在 HTML 中, 也可以嵌套在 XML 中。使用 ZPT 创建模板, 可以使用 Dreamweaver 这样的所见即所得的编辑器创建模板。然后可以向模板中添加 TAL, 添加了 TAL 的模板仍可以被 Dreamweaver 修改而不影响。ZAL 常用的标记有以下几种。

- tal:attributes: 修改属性。

- tal:condition: 条件测试。
- tal:content: 输出内容。
- tal:define: 定义变量。
- tal:on-error: 异常处理。
- tal:repeat: 循环。
- tal:replace: 替换内容。

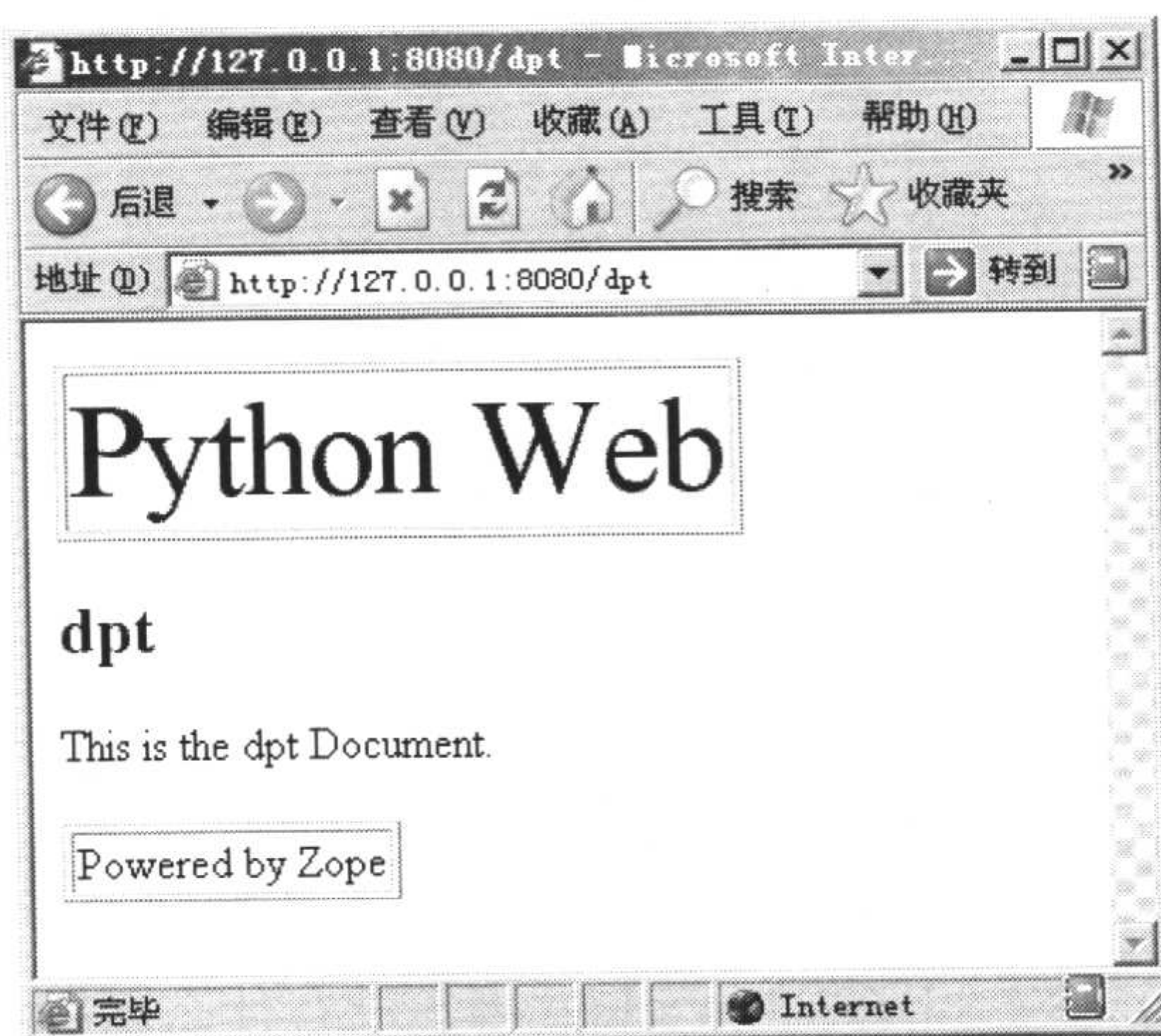


图 17-16 使用 DTML 创建模板

在 Zope 中添加一个 Id 为 “zpt” 的 Page Template，修改其内容如下所示。

```
<html>
  <head>
    <title tal:content="template/title">The title</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
  </head>
  <body>
<table border="1" width="100%">
  <tr>
    <th>Id</th>
  </tr>
  <tr tal:repeat="item context/objectValues">
    <td tal:content="item/getId">Id</td>
  </tr>
</table>
  </body>
</html>
```

保存 “zpt” 后，在 IE 地址栏中输入 `http://127.0.0.1:8080/zpt`，如图 17-17 所示。

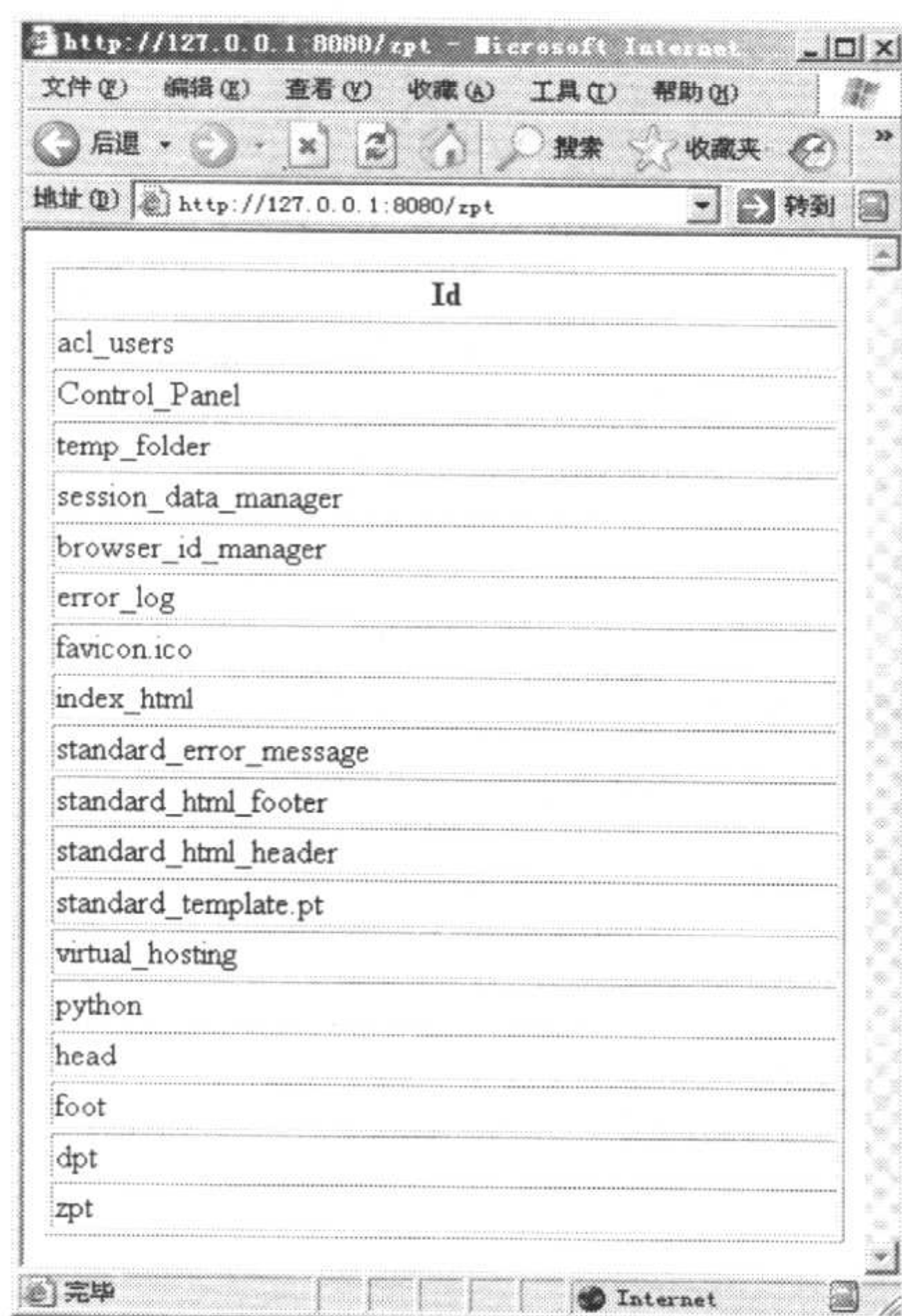


图 17-17 使用 ZPT 创建模板

17.1.4 添加 Python 脚本

在 Zope 中使用 Python 脚本和普通的 Python 脚本没有什么区别,不过,为了提高安全性,Zope 对 Python 脚本做了限制。例如,不能使用内置 open 函数,range 函数不能产生较大的循环等。在 Zope 中表单所提交的内容也可以通过参数的形式传递给 Python 脚本。在 Zope 中添加一个 Id 为“submit”的 DTML Document 文件,其内容如下所示。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Input your name and age</title>
</head>

<body>
<form id="form1" name="form1" method="post" action="show">
  <label>Name
  <input type="text" name="name" />
</label>
<p>
  <label>Age
  <input type="text" name="age" />
  </label>
</p>
<p>
  <input type="submit" name="Submit" value="提交" />
</p>
</body>
</html>
```



```



```

在 Zope 中新添加一个 Id 为 “show” 的 Script (Python) 脚本, 在其【Parameter List】文本框中输入 “name,age”, 如 “Tom, 20”, 即表单所提交的变量名。修改其内容如下所示。

```

print "<center>"
print "<p>Your name is:</p>"
print "<p>", name, "</p>"
print "<p>Your age is:</p>"
print "<p>", age, "</p>"
print "</center>"
return printed

```

在 IE 地址栏中输入 <http://127.0.0.1:8080/submit>, 在【Name】文本框中输入 “Tom”, 在【Age】文本框中输入 “20”, 如图 17-18 所示。单击【提交】按钮后, 如图 17-19 所示。

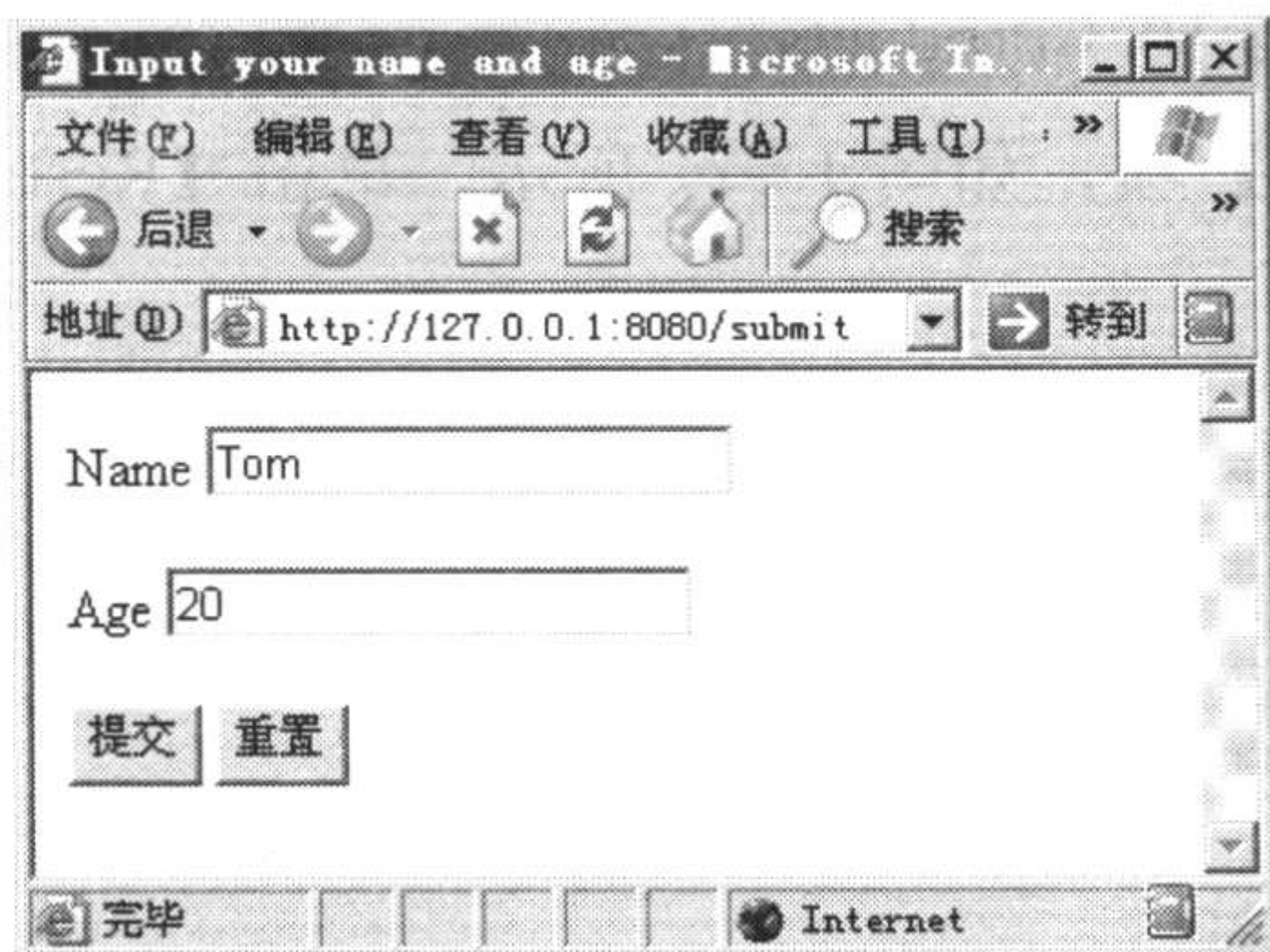


图 17-18 提交表单

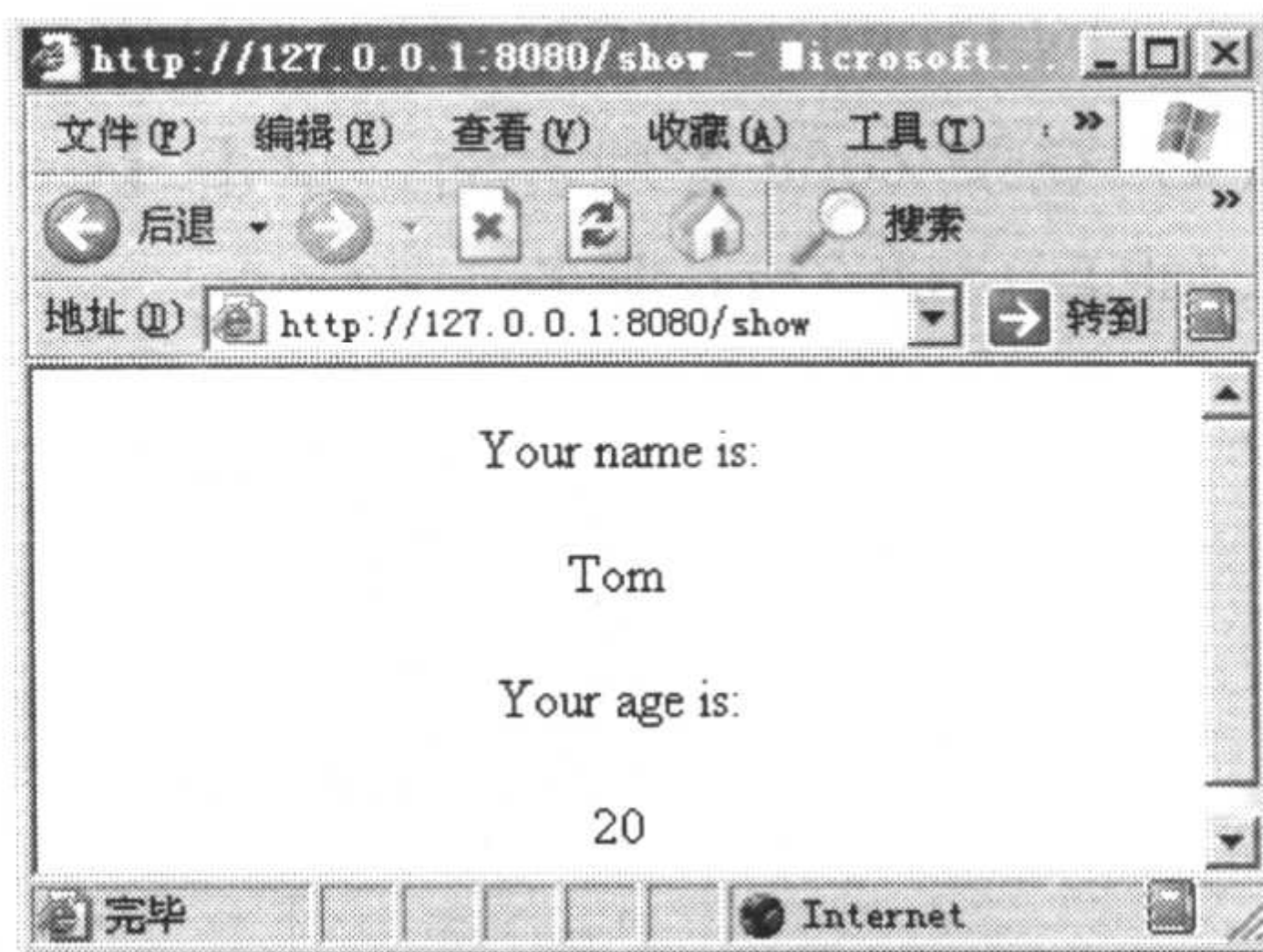


图 17-19 Python 脚本输出

17.1.5 连接 MySQL 数据库

在 Zope 中集成了简单的 Gadfly 数据库, 可以完成基本的存储任务。在 Zope 中还可以使用 MySQL 数据库, 因为 Zope 官方提供的某些安装包已经自带了 Python 程序, 这就需要为 Zope 自带的 Python 程序安装 MySQLdb 模块。以 Zope-2.10.3 为例, 配置过程如下。

(1) 单击【开始】|【所有程序】|【Zope 2.10.3】|【Python】|【Set InstallPath】命令, 将 Zope 自带的 Python 路径添加到注册表中。

(2) 从 MySQLdb 官方网站 <http://sourceforge.net/projects/mysql-python> 下载 MySQL-python-1.2.2.win32-py2.4.exe 安装程序, 双击安装程序进入安装界面。单击【下一步】按钮, 将出现如图 17-20 所示的界面, 表示安装程序找到 Python 安装路径, 则只需安装默认安装即可。如果未出现如图 17-20 所示的界面, 表示 Zope 自带的 Python 安装路径未添加到注册表, 需要重新运行步骤 (1)。

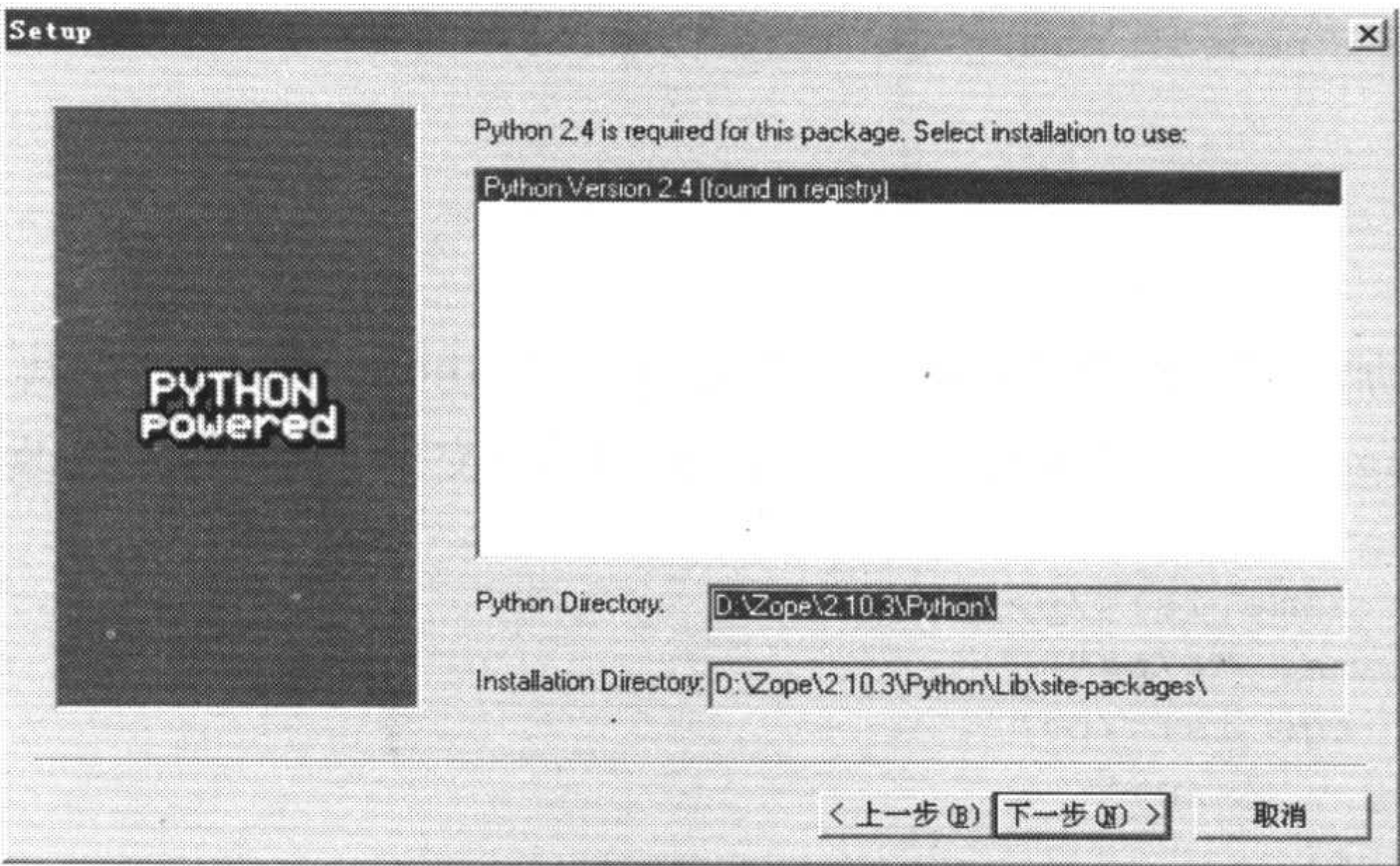


图 17-20 安装路径

(3) 从 MySQLdb 官方网站 <http://sourceforge.net/projects/mysql-python> 下载 ZMySQLDA-2.0.8.tar.gz, 将该压缩包中 lib\python\Products 目录下的 ZMySQLDA 目录复制到 Zope 服务安装目录下的 Products 目录, 例如 “D:\Zope\Instance\2.10.3\Products”。

(4) 登录 Zope 管理界面, 单击 **【Control Panel】** 项, 如图 17-21 所示。单击 **【Restart】** 按钮, 重新启动 Zope 服务。

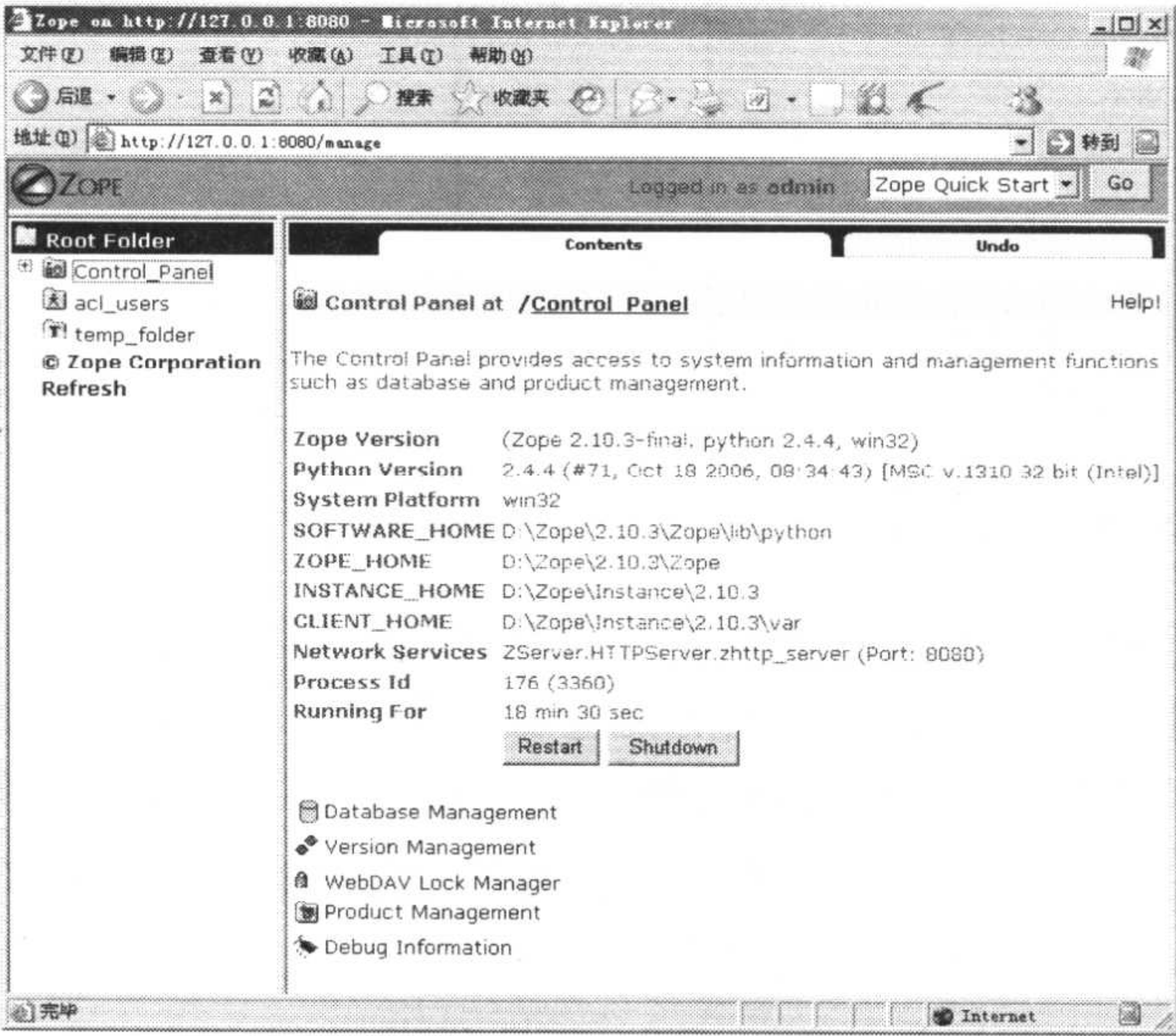


图 17-21 重新启动 Zope 服务

(5) Zope 服务重新启动后, 选中 **【Root Folder】** 选项, 单击 **【Select type to add】** 下拉列表框, 可以看到 **【Z MySQL Database Connection】** 项, 如图 17-22 所示, 此时即可在 Zope 中使用 MySQL 数据。

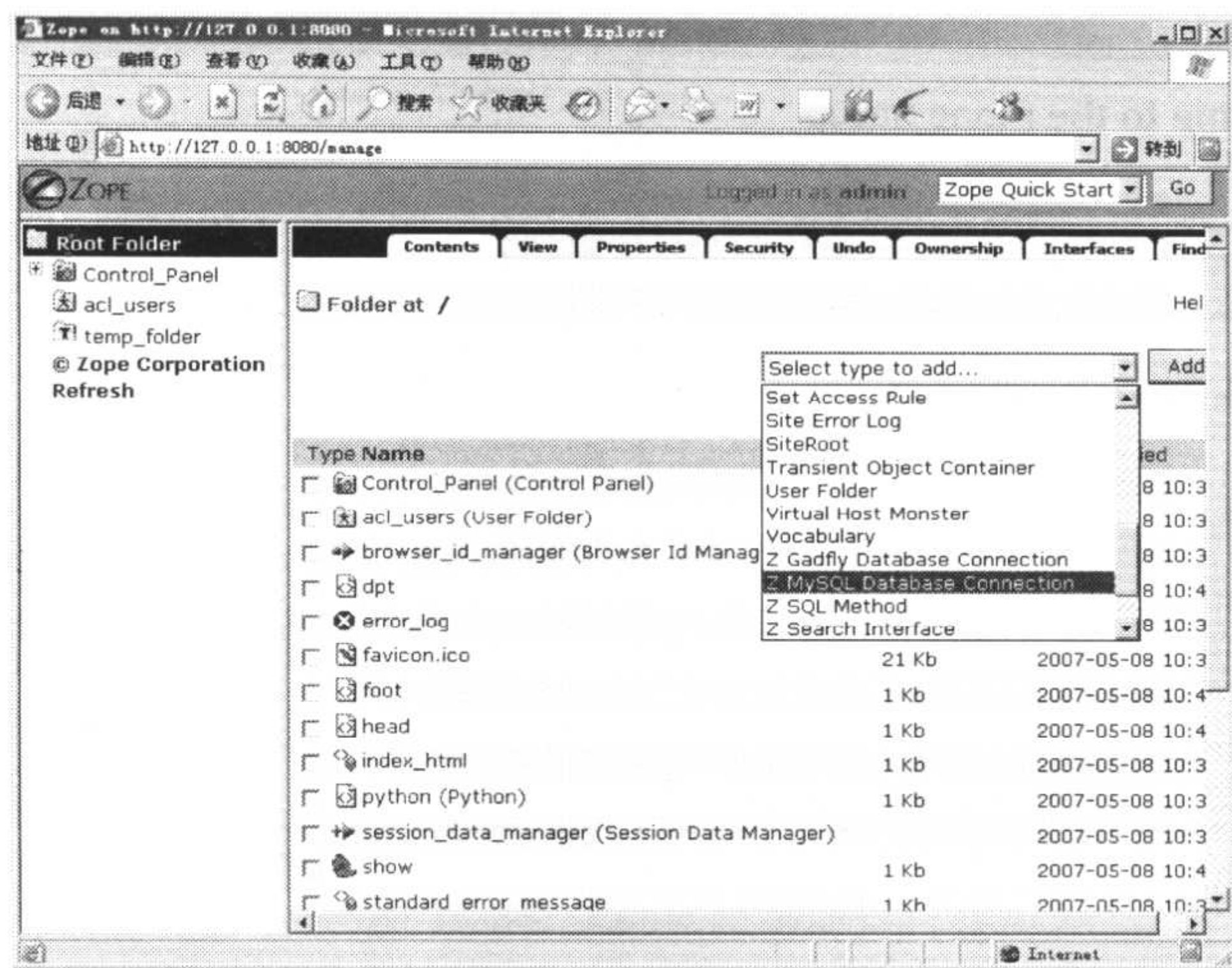


图 17-22 ZMySQL Database Connection

17.2 使用 Plone 内容管理系统

Plone 是基于 Zope 的内容管理系统，使用 Plone 可以建立网站、发布系统等。Zope 的使用过于复杂，对于初学者而言，在 Zope 下完整地建立一个网站需要花费不少时间。Plone 使得建立网站变得容易起来。对于企业内部，Plone 可以作为企业内部网的服务器，发布通知、文档等。Plone 还提供了 Blog、论坛、Wiki 等扩展产品，为用户提供了丰富的选择。

17.2.1 安装 Plone

Plone 提供了 Windows 下的完整安装程序，其中包含了 Zope。该安装程序适合初学者使用，避免了复杂的配置。如果已经安装了 Zope，可以在 Zope 下安装 Plone。为了方便起见，应卸载 Zope，重新安装带有 Zope 的 Plone。以 plone-2.5.2-zope-2.9 为例，Plone 在 Windows 下的安装步骤如下。

(1) 双击运行安装程序，如图 17-23 所示。

(2) 单击【Next】按钮，进入安装协议界面，选中【I accept the agreement】单选框，如图 17-24 所示。

(3) 单击【Next】按钮，进入安装路径选择界面，此处可以根据需要选择 Plone 的安装路径，如图 17-25 所示。

(4) 单击【Next】按钮，进入设置管理员用户名和密码的界面，此处设置 Zope 管理界面用户名和密码，以及 Plone 上的用户名和密码，如图 17-26 所示。

(5) 单击【Next】按钮，进入安装确认界面，如图 17-27 所示，单击【Install】按钮，开始安装 Plone。

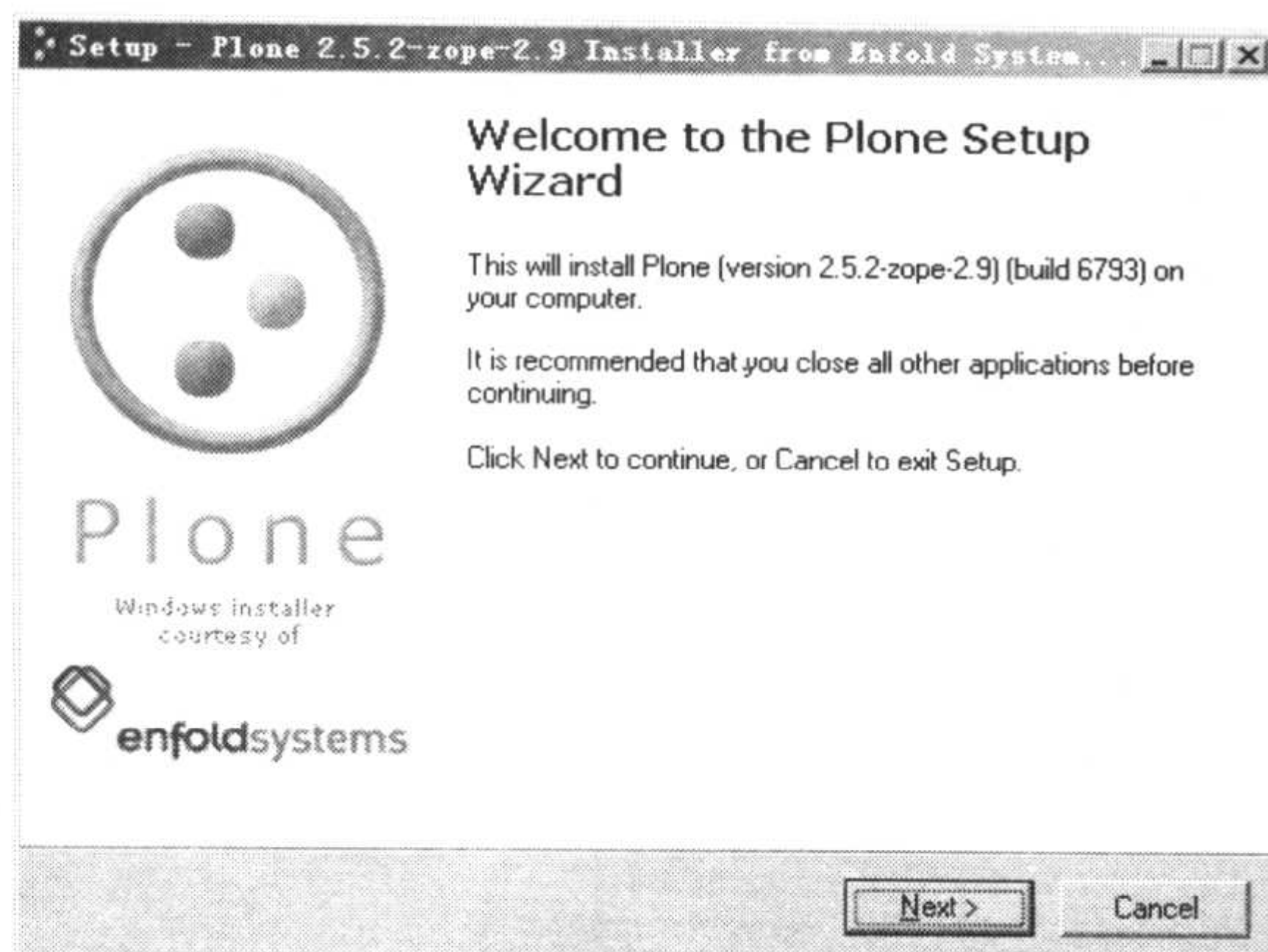


图 17-23 Plone 安装程序

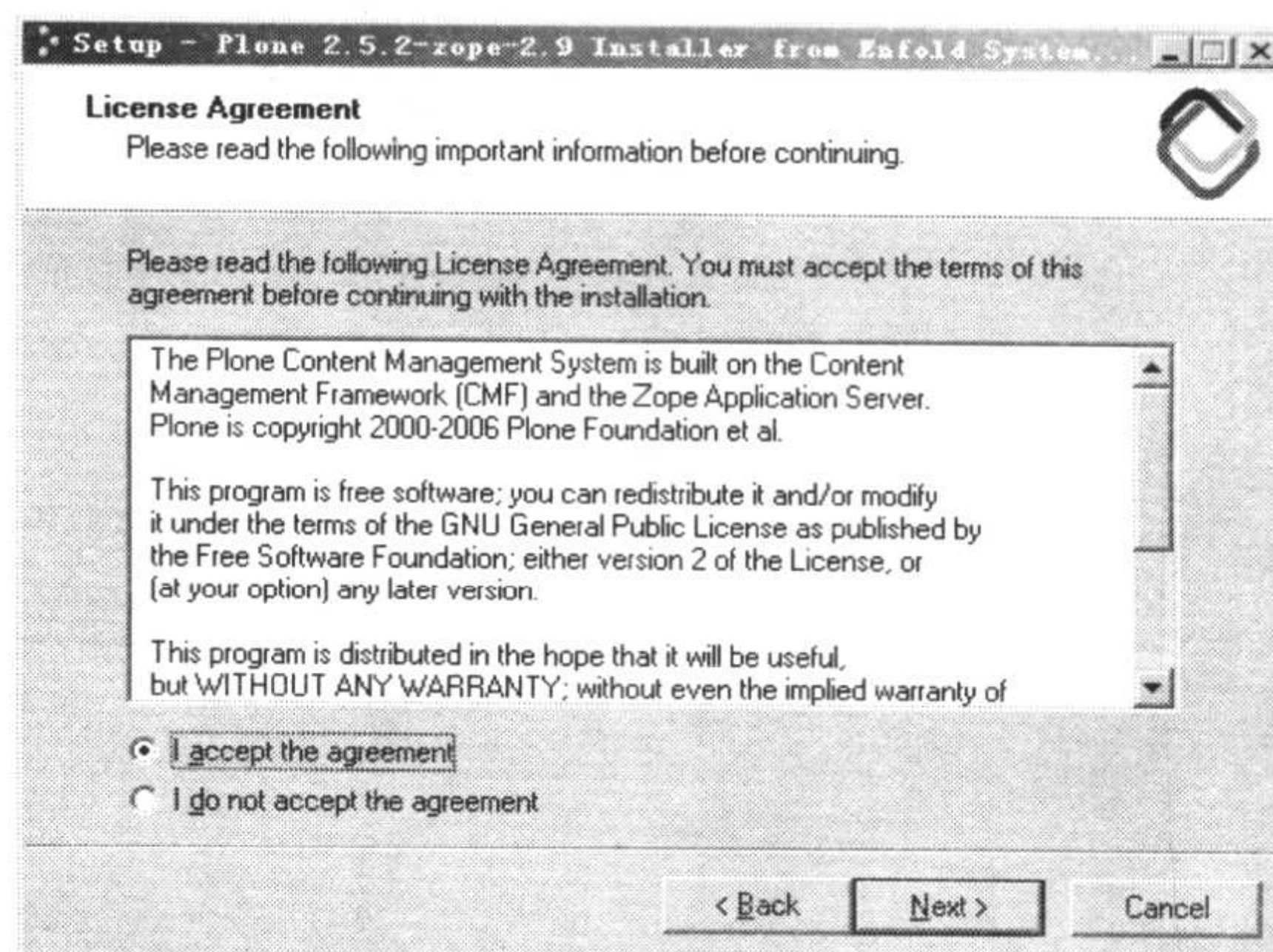


图 17-24 安装协议

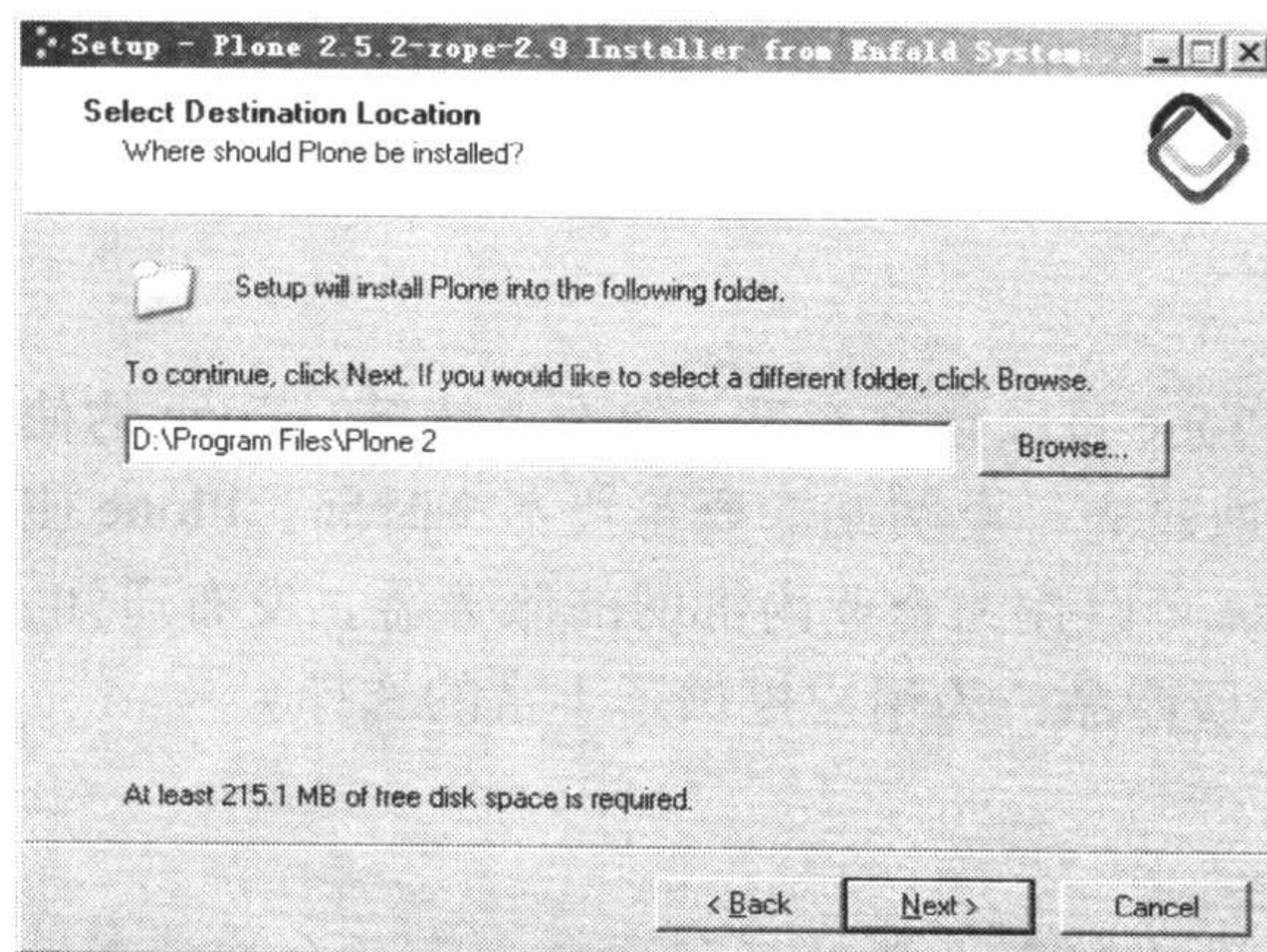


图 17-25 安装路径

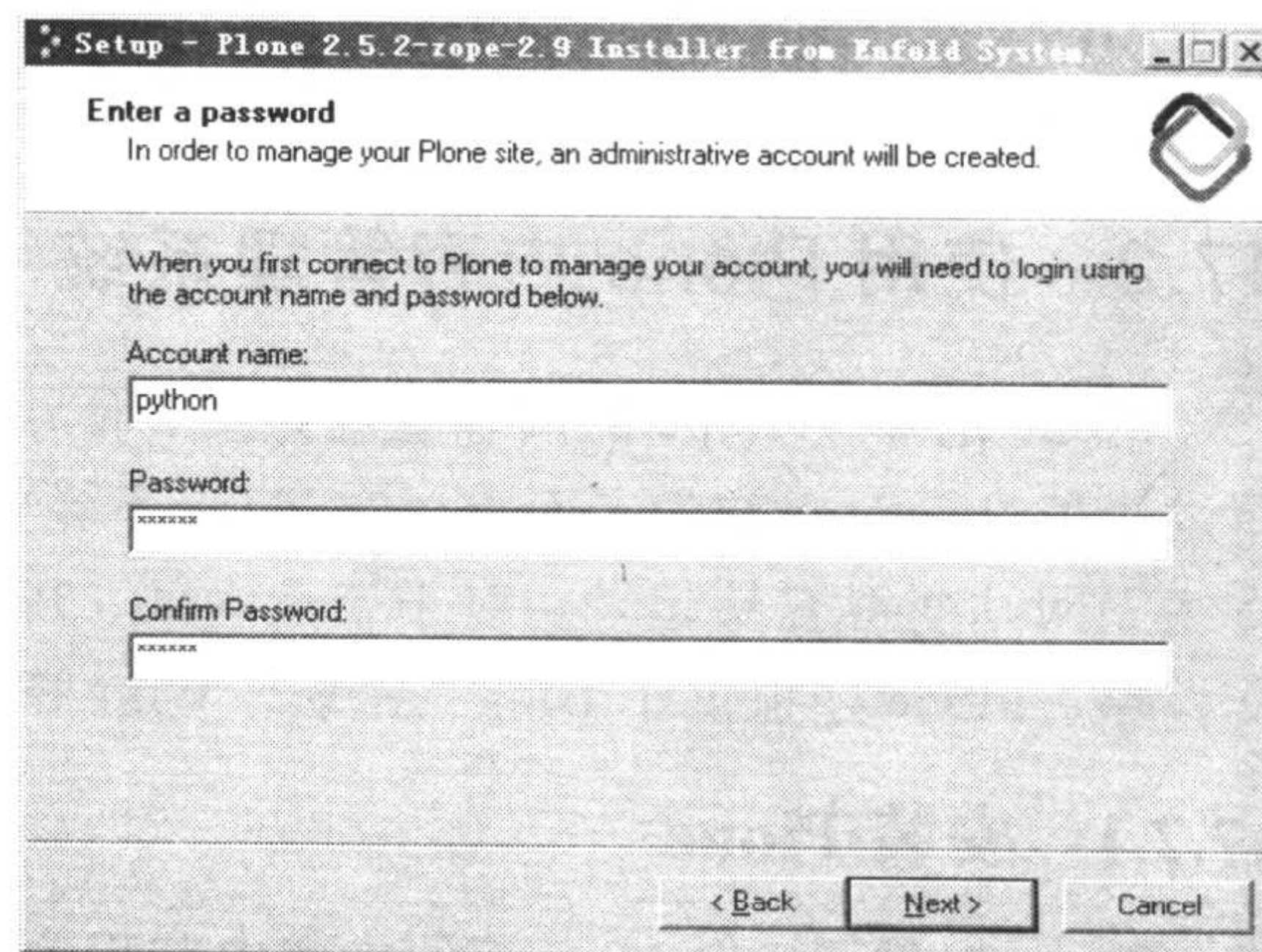


图 17-26 管理员名和密码

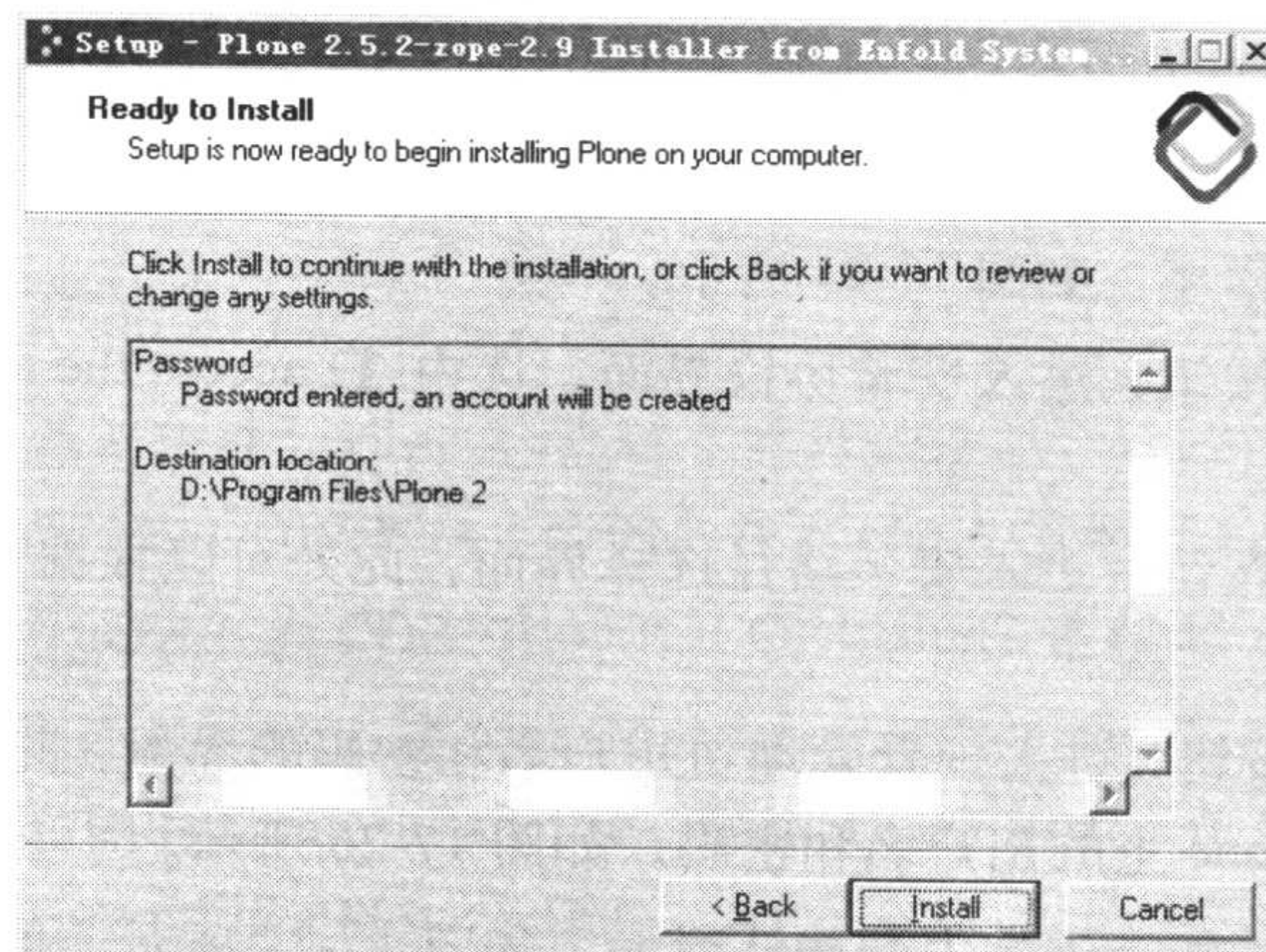


图 17-27 确认安装

(6) 安装完成后, 单击【开始】|【所有程序】|【Plone】|【Plone】命令, 运行 Plone 控制面板, 如图 17-28 所示。

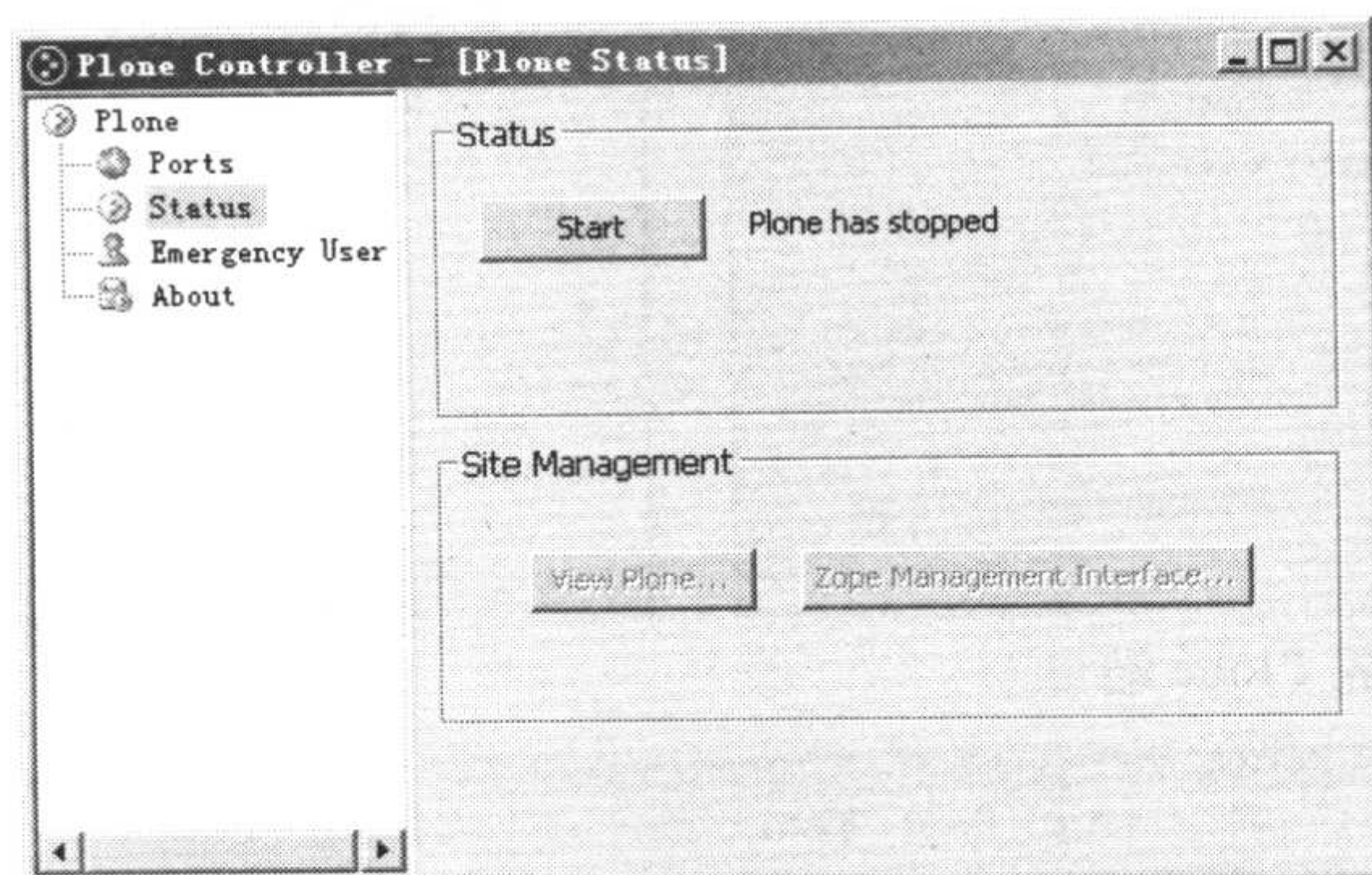


图 17-28 Plone 控制面板

(7) 单击【Start】按钮, 开启 Plone, 单击【View Plone】按钮, 可以打开 Plone 界面, 如图 17-29 所示。

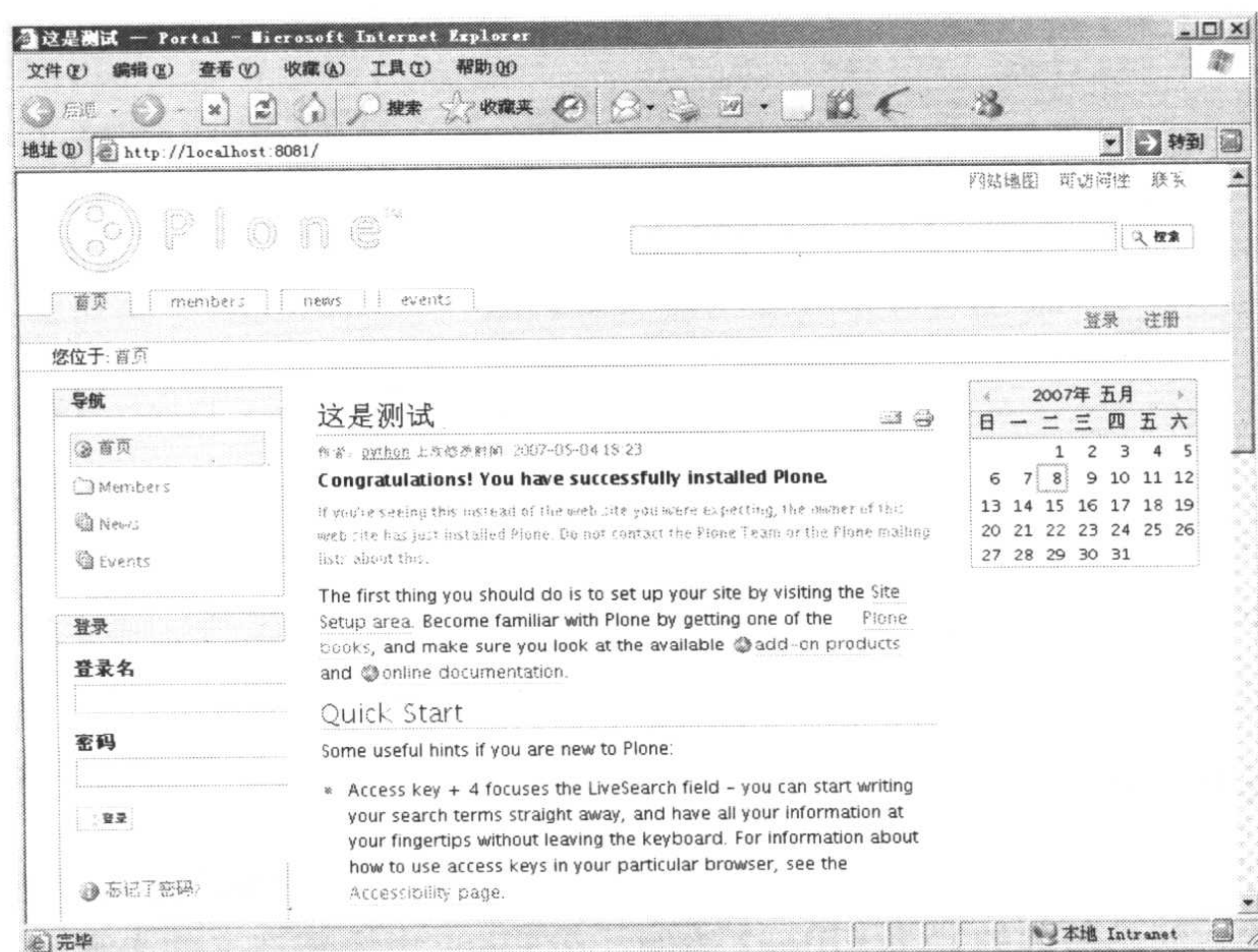


图 17-29 Plone 界面

(8) 单击 Plone 控制面板右侧列表中的【Ports】项可以更改 Plone 的端口, 如图 17-30 所示。单击【Emergency User】选项可以创建新用户, 如图 17-31 所示。

(9) 在 Plone 的 Web 界面使用安装时创建的用户名和密码可以登录到 Plone 系统中, 进行内容添加、修改等操作, 如图 17-32 所示。

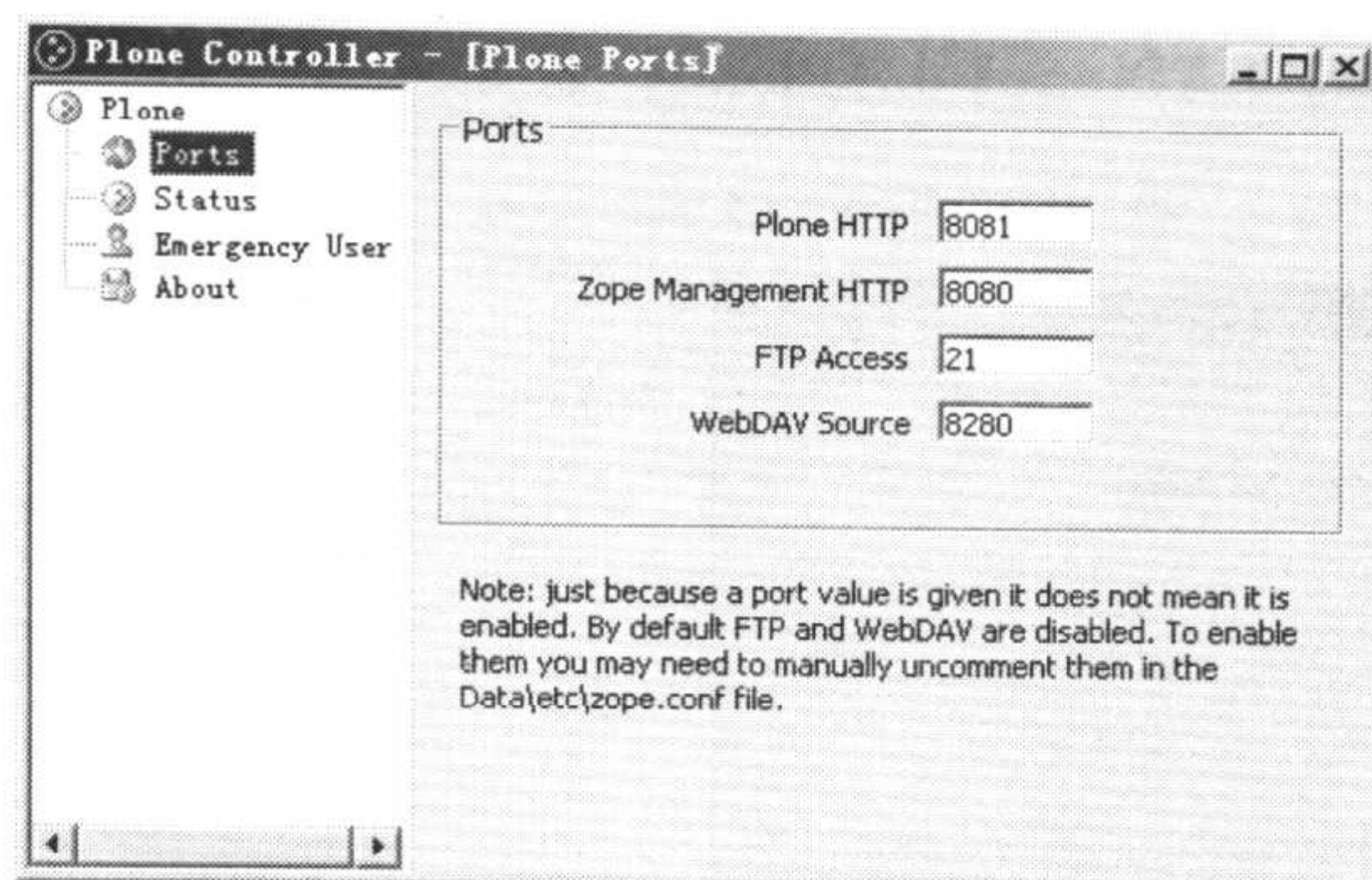


图 17-30 设置 Plone 端口

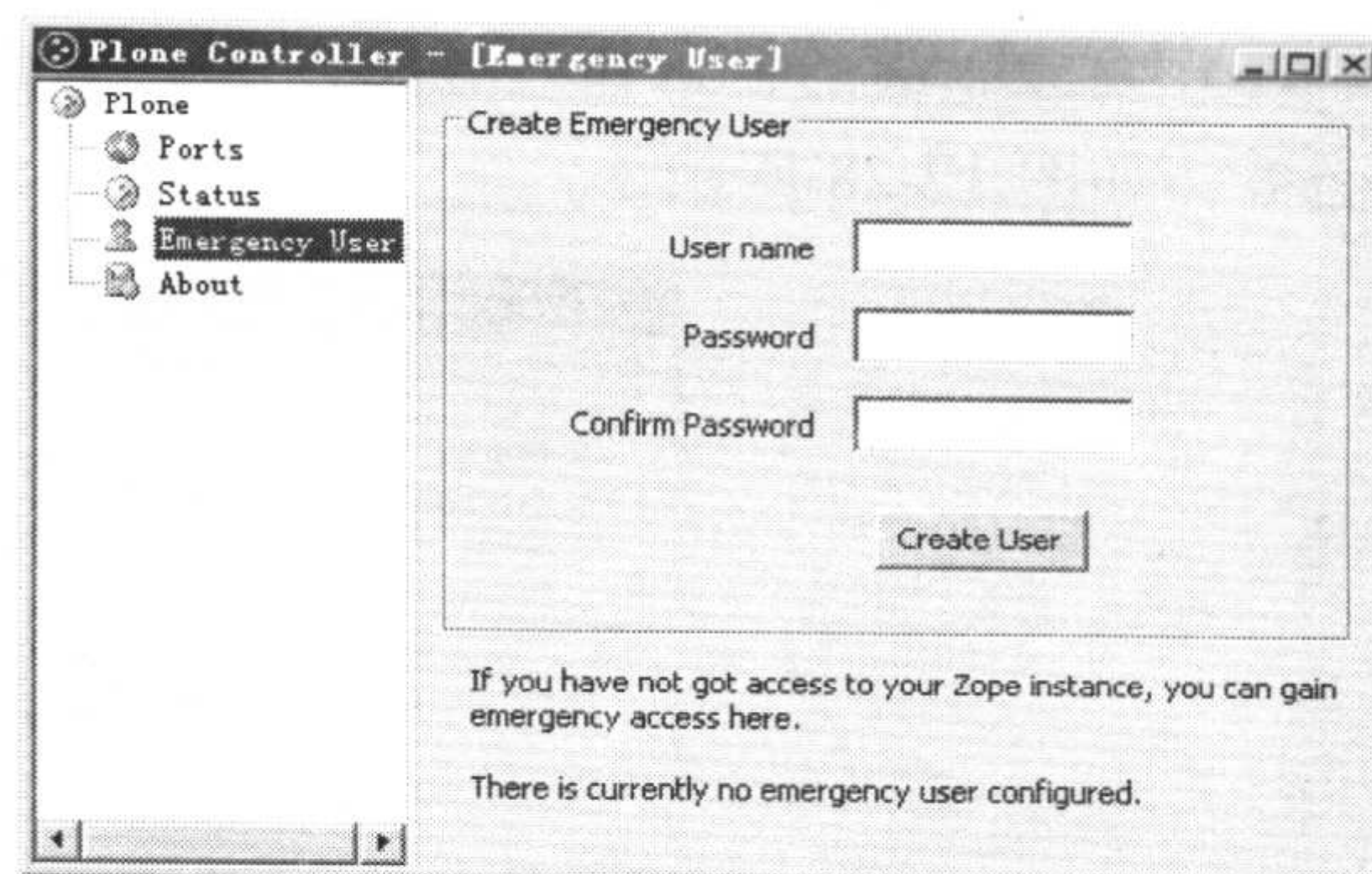


图 17-31 创建用户

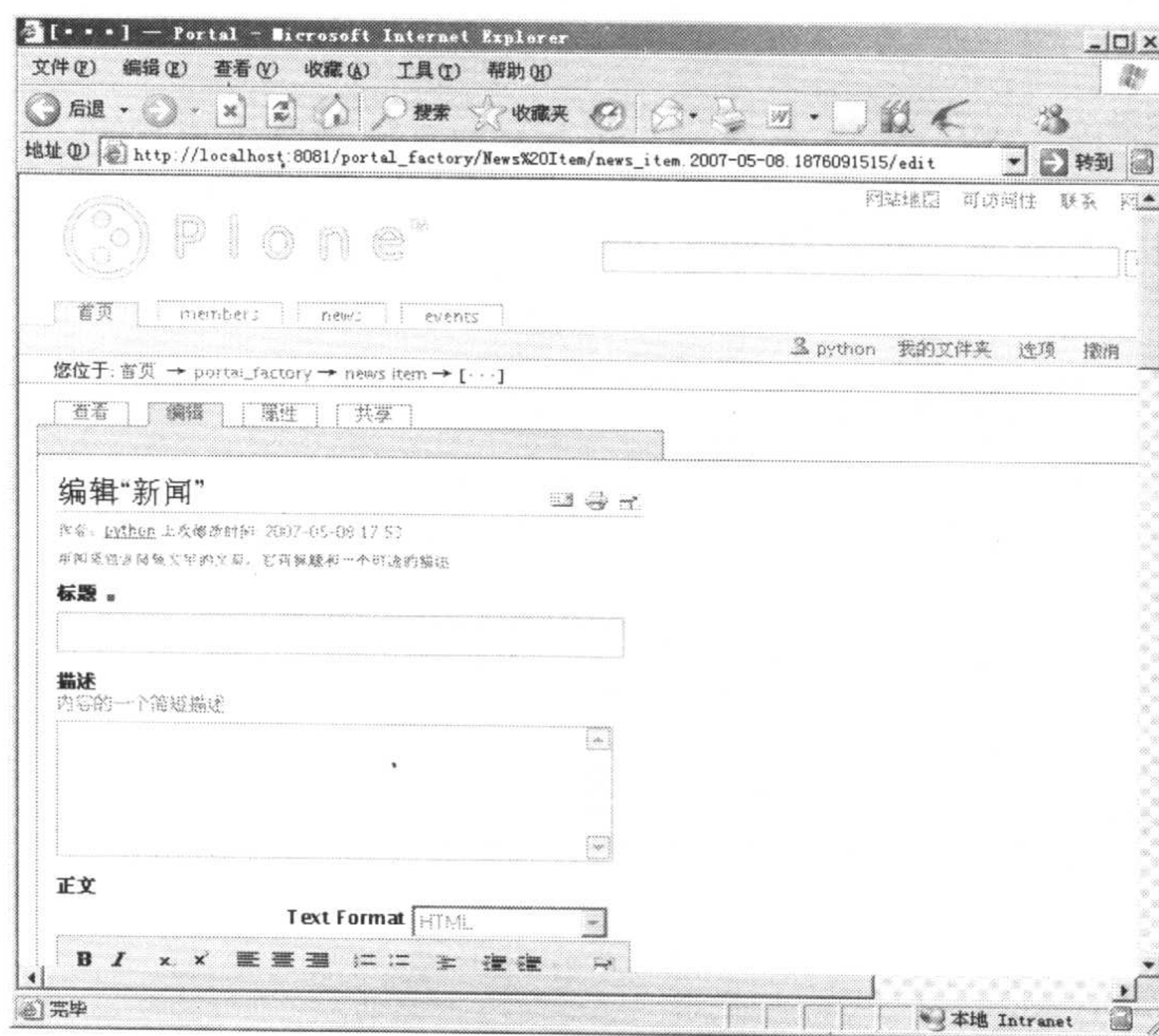


图 17-32 在 Plone 中添加新闻

17.2.2 安装 Plone 产品

除了基本的内容管理系统外，Plone 还提供了其他的产品，这些产品可以被安装到 Plone 中。以 Plone 官方网站提供的 COREBlog2 为例，在 Plone 中安装产品的步骤如下。

- (1) 从 Plone 的官方网站下载 COREBlog2。COREBlog2 是在 Plone 下创建 Blog 的产品。
- (2) 运行 Plone 控制面板，单击【Zope Management Interface】按钮，进入 Zope 的管理界面，单击管理界面中右侧的 Control Panel 项，如图 17-33 所示。查看“INSTANCE_HOME”的路径，此处为“d:\Program Files\Plone 2\Data”。
- (3) 将 COREBlog2 压缩包中的“COREBlog2”目录解压到“d:\Program Files\Plone 2\Data\”。

Products” 目录中。

(4) 重新启动 Zope 服务后, 登录 Plone 界面, 单击窗口右上角的“网站设置”链接, 如图 17-34 所示。

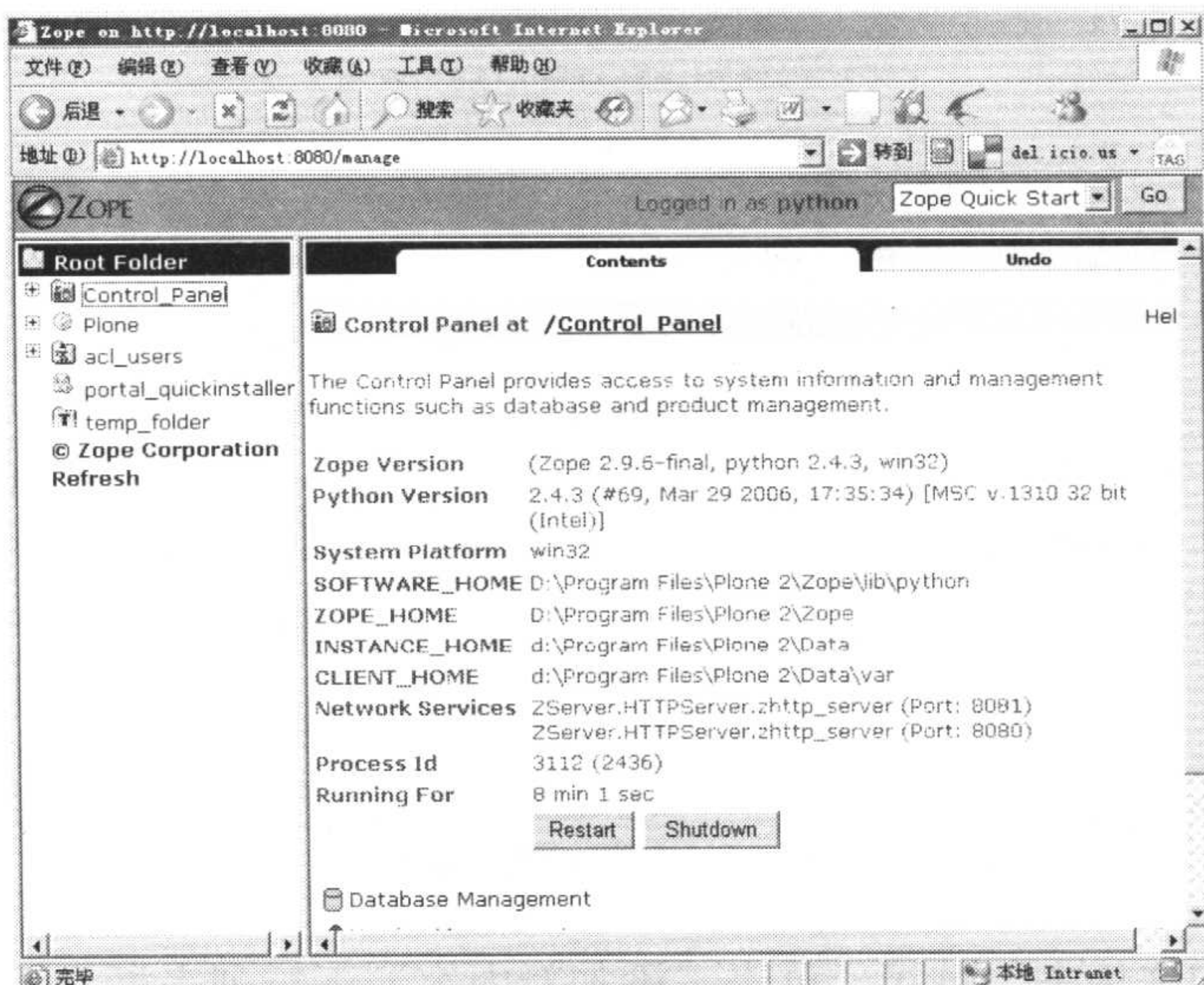


图 17-33 查看 INSTANCE_HOME 的路径



图 17-34 网站设置界面

(5) 单击“安装/卸载产品”链接, 如图 17-35 所示。选中“COREBlog2 9.82b”复选框, 单击【安装】按钮将安装 COREBlog2。

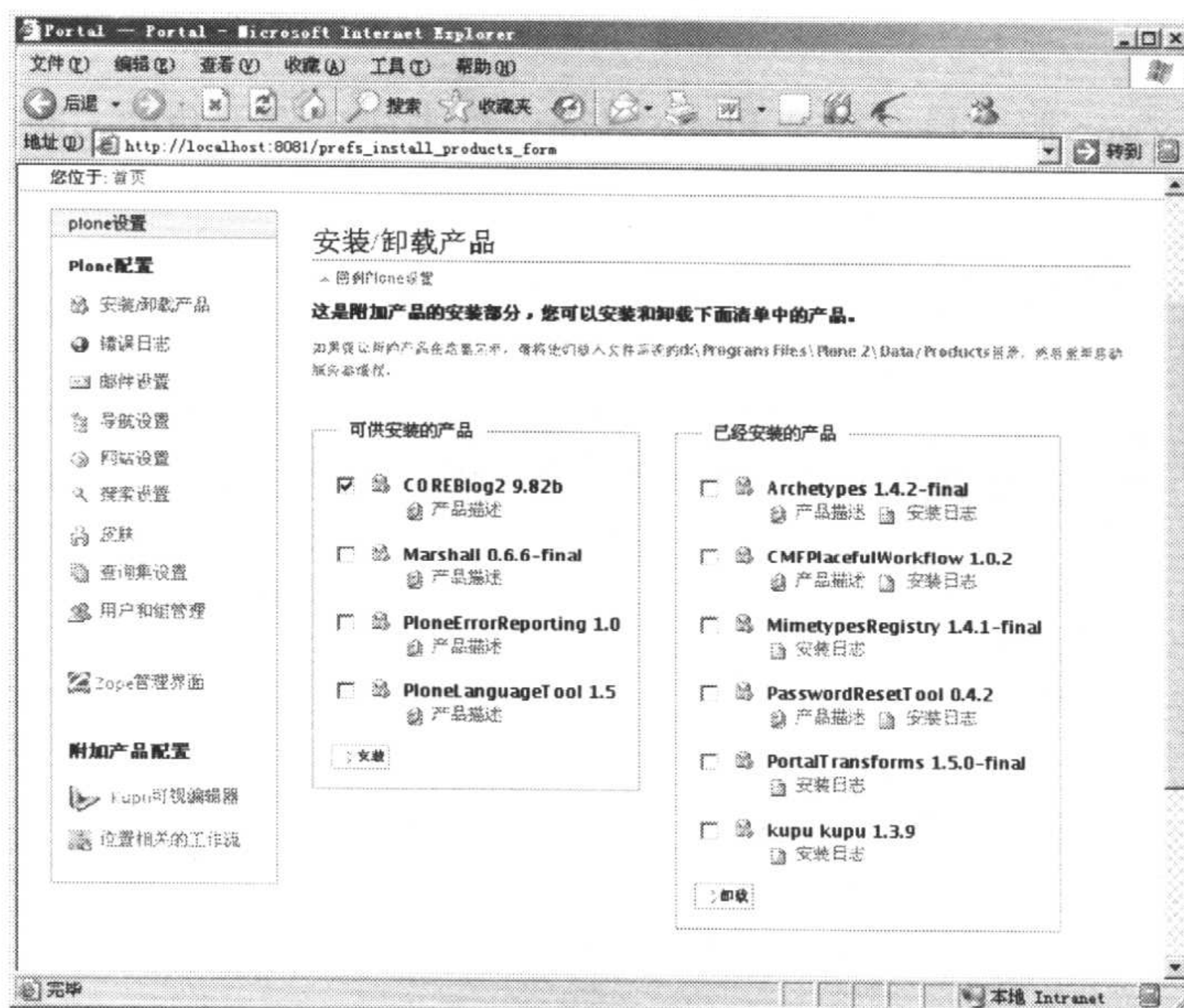


图 17-35 安装 COREBlog2

(6) 单击“我的文件夹”链接，单击“添加新内容”下拉列表框，如图 17-36 所示，选择“coreblog2”选项，如图 17-37 所示，在 ID 中填入“Blog”，直接单击【保存】按钮，按照默认设置创建 Blog。



图 17-36 添加 coreblog2



图 17-37 设置 COREBlog2

(7) 添加 COREBlog 后，应首先添加一个 Categories，单击“Categories”链接，选择“添加新内容”下拉列表框中的“category”项，如图 17-38 所示。按照如图 17-39 所示的内容进行填写，即可完成添加 Categories。

(8) 添加完 Categories 后就可以单击“Entries”链接向 Blog 中添加文章。

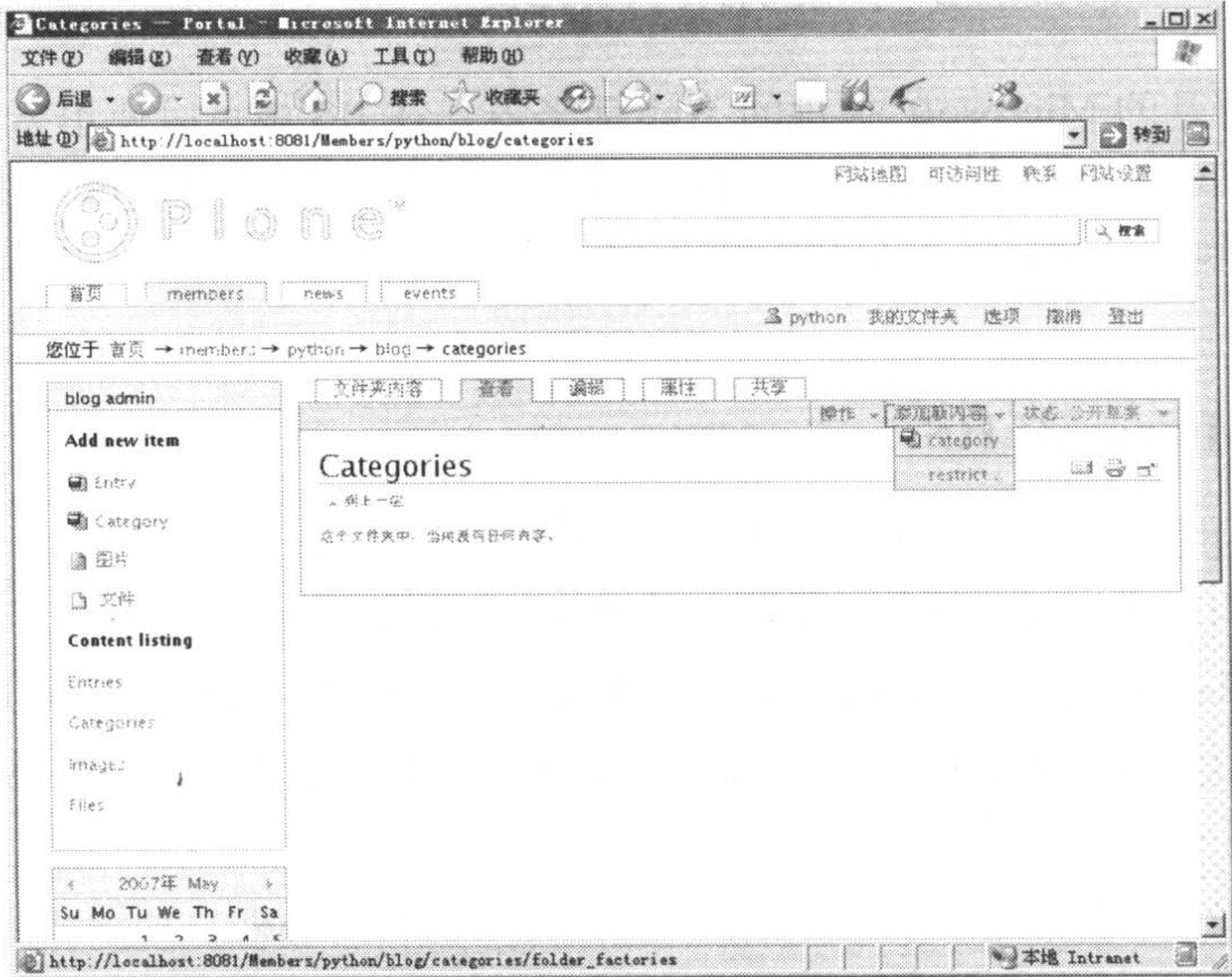


图 17-38 添加 Categories

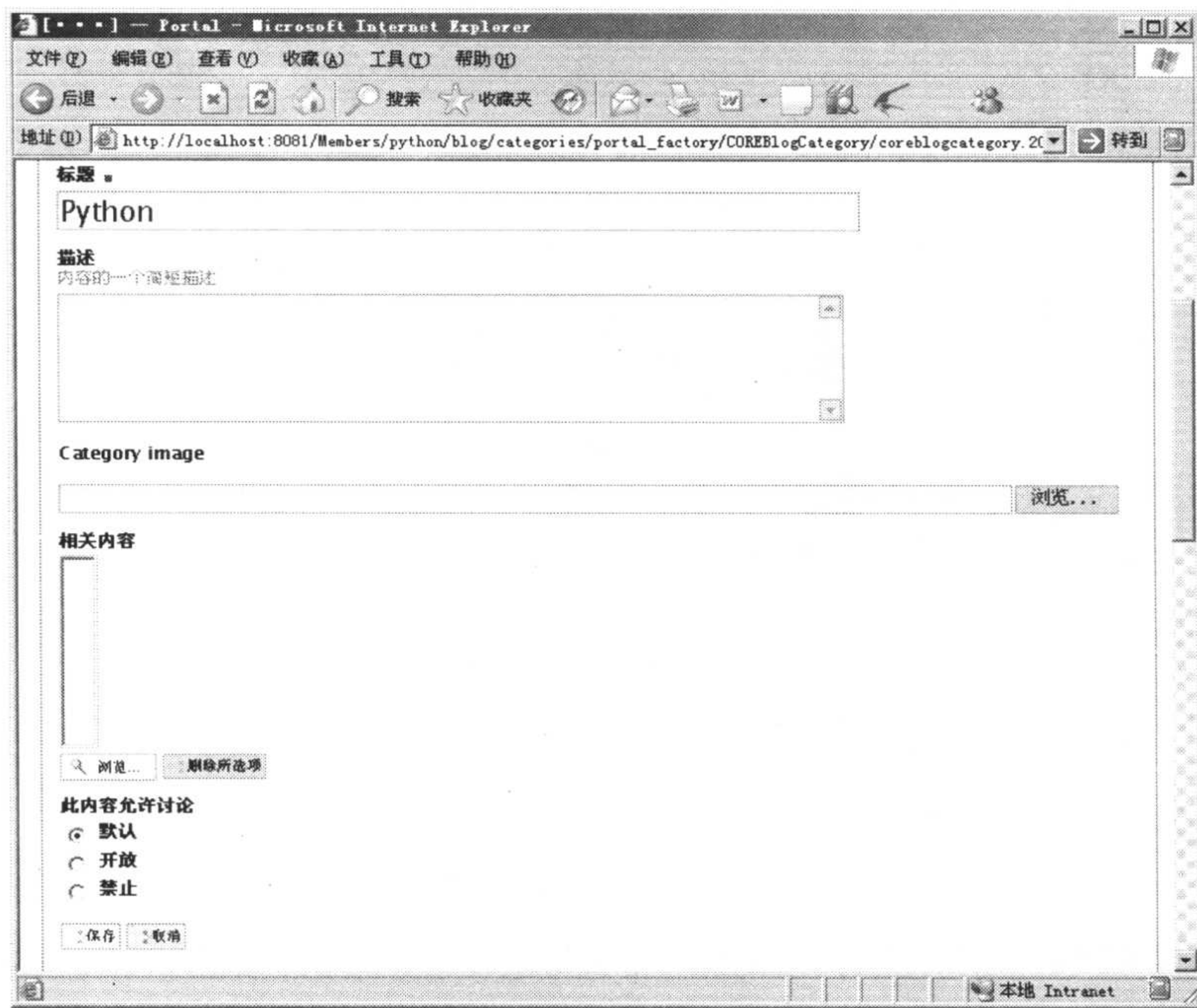


图 17-39 设置 Categories

17.3 在 Microsoft IIS 中使用 Python

Microsoft IIS 是的 Microsoft 提供的 Web 服务器。在 IIS 中可以使用 ASP (Active Server Pages) 创建动态网站。ASP 本身并不是脚本语言，但在 ASP 中可以嵌入其他的脚本语言，例如 VBscript、Javascript 和 Python。也可以直接在 IIS 中使用 Python 脚本代替“.asp”文件。

17.3.1 安装 Microsoft IIS

在 Windows 下需要安装 Microsoft IIS 才能支持 ASP。Microsoft IIS 作为 Windows 的组件，默认是不被安装的。在 Windows XP 操作系统下安装 IIS 的步骤如下所示。

- (1) 单击【开始】|【控制面板】|【添加/删除程序】命令，如图 17-40 所示。
- (2) 单击【添加/删除 Windows 组件】按钮，如图 17-41 所示，选中 Internet 信息服务 (IIS)，单击【下一步】按钮。
- (3) 安装过程中将弹出插入 Windows XP 安装光盘的消息框，如图 17-42 所示，此时将 Windows XP 的安装盘插入光驱后，单击【确定】按钮。

第17章 Python Web 应用

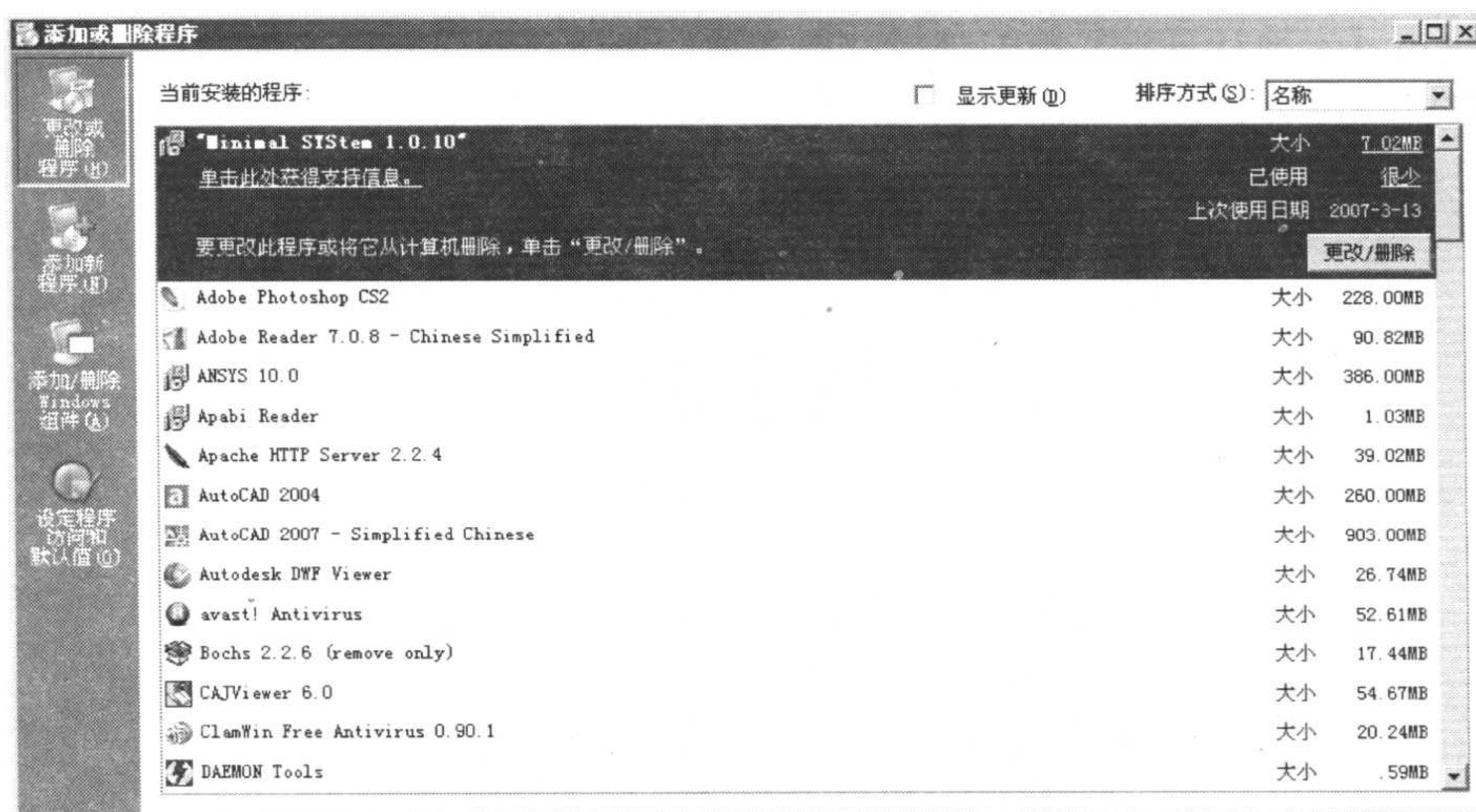


图 17-40 添加/删除程序

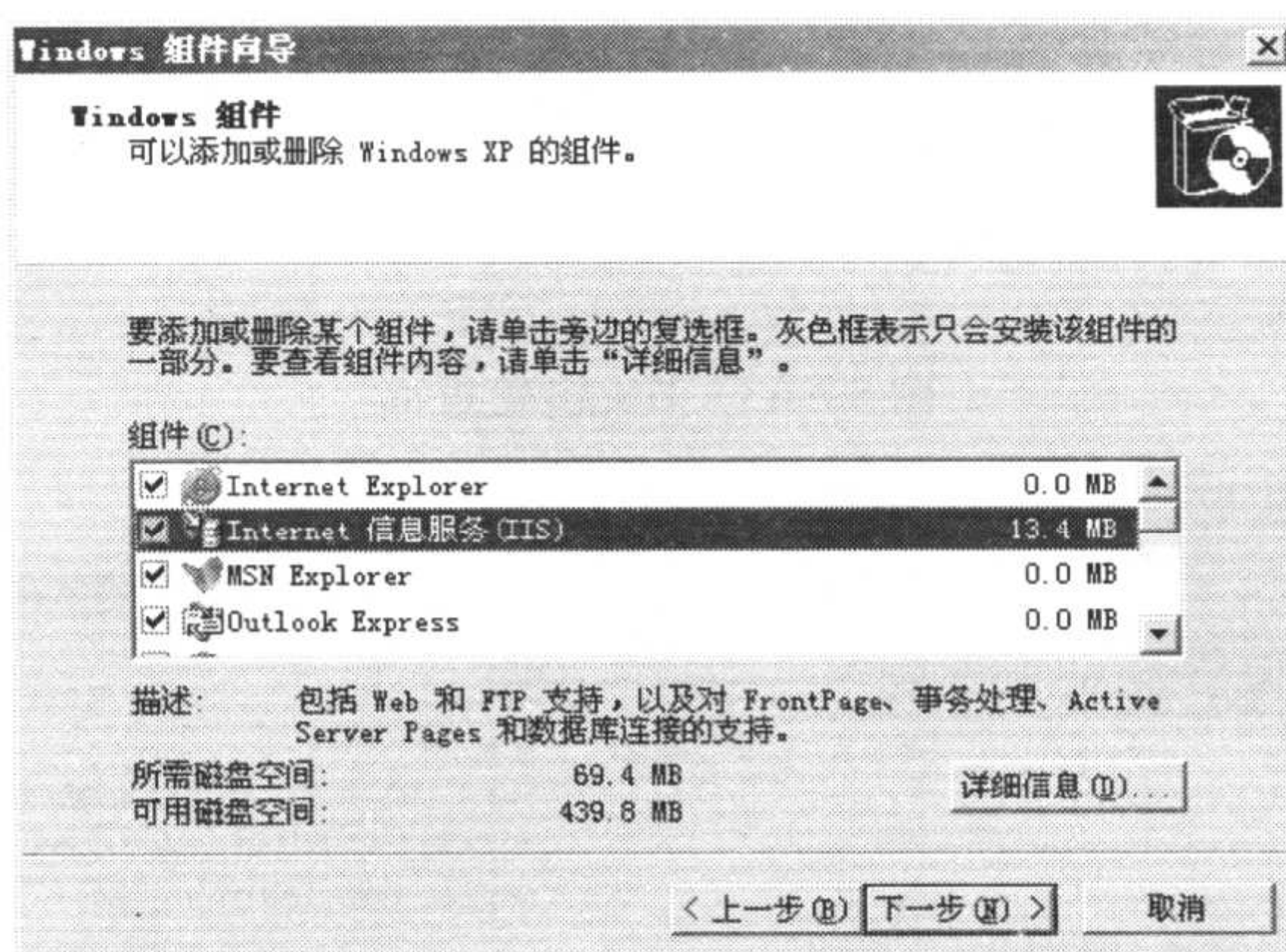


图 17-41 安装 IIS

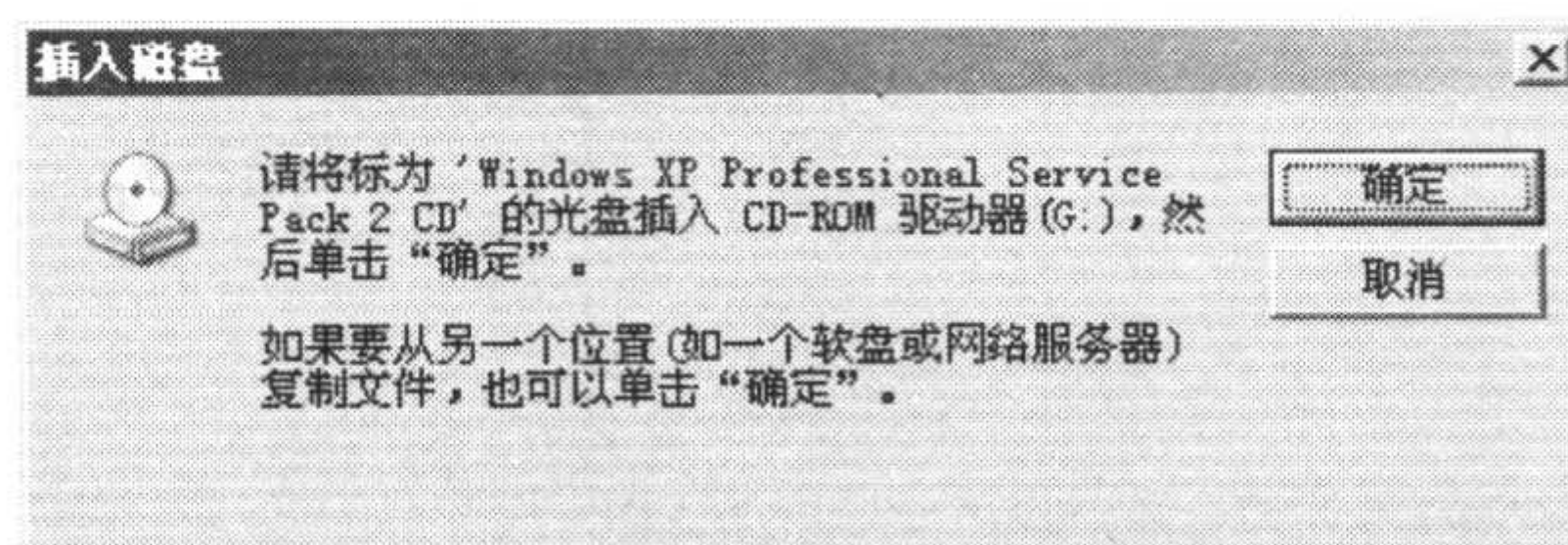


图 17-42 提示插入安装盘

(4) 安装完成后，在 IE 地址栏中输入 `http://localhost/localstart.asp`，将打开如图 17-43 所示的界面，表示 IIS 已经安装成功。

(5) IIS 安装完成后，单击【开始】|【控制面板】|【管理工具】|【Internet 信息服务】命令，如图 17-44 所示。

(6) 单击【本地计算机】前的加号，展开项，单击【网站】前的加号，在【默认网站】项上右击选择【属性】命令，如图 17-45 所示。

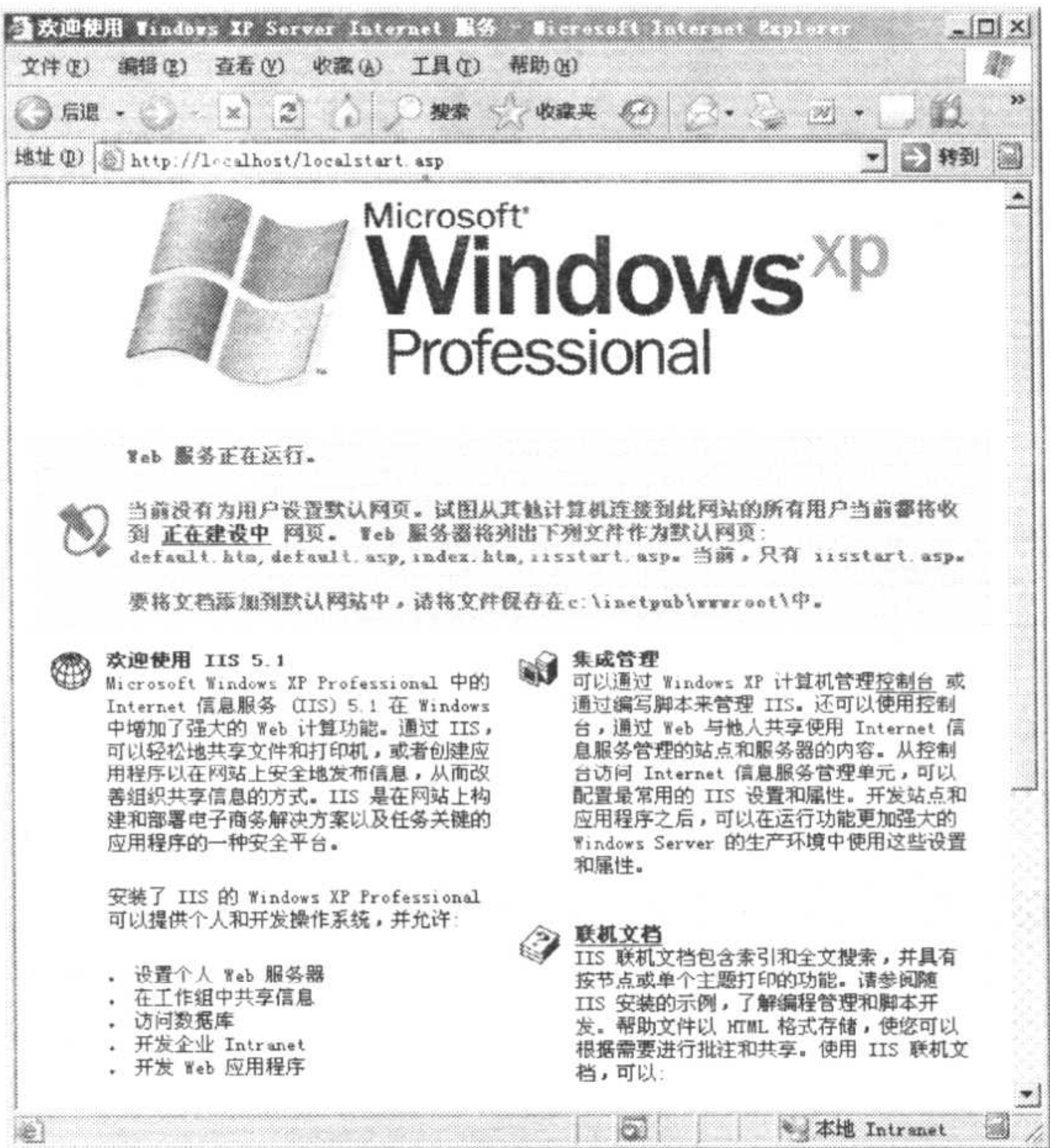


图 17-43 IIS 起始页面

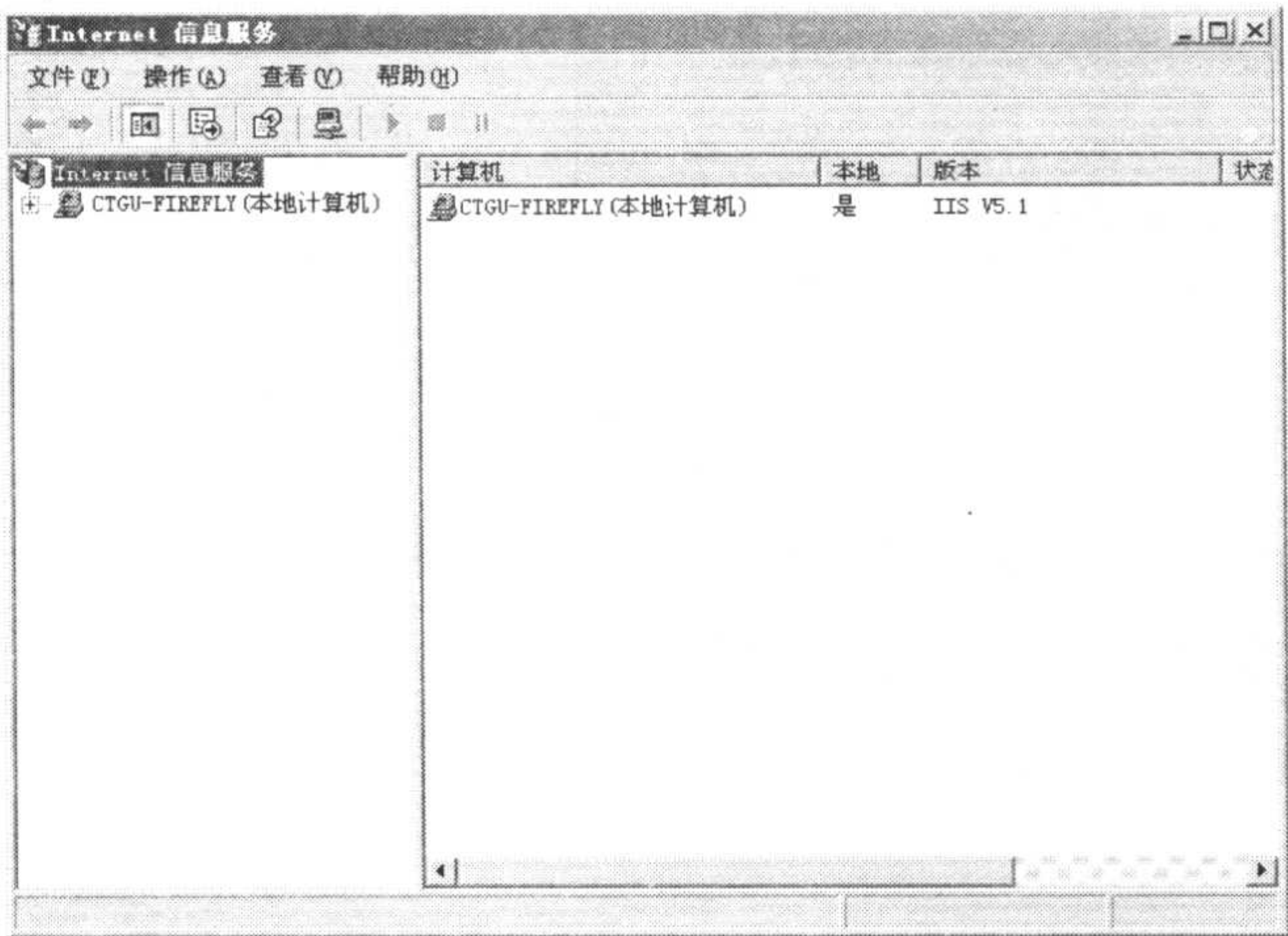


图 17-44 IIS 管理界面

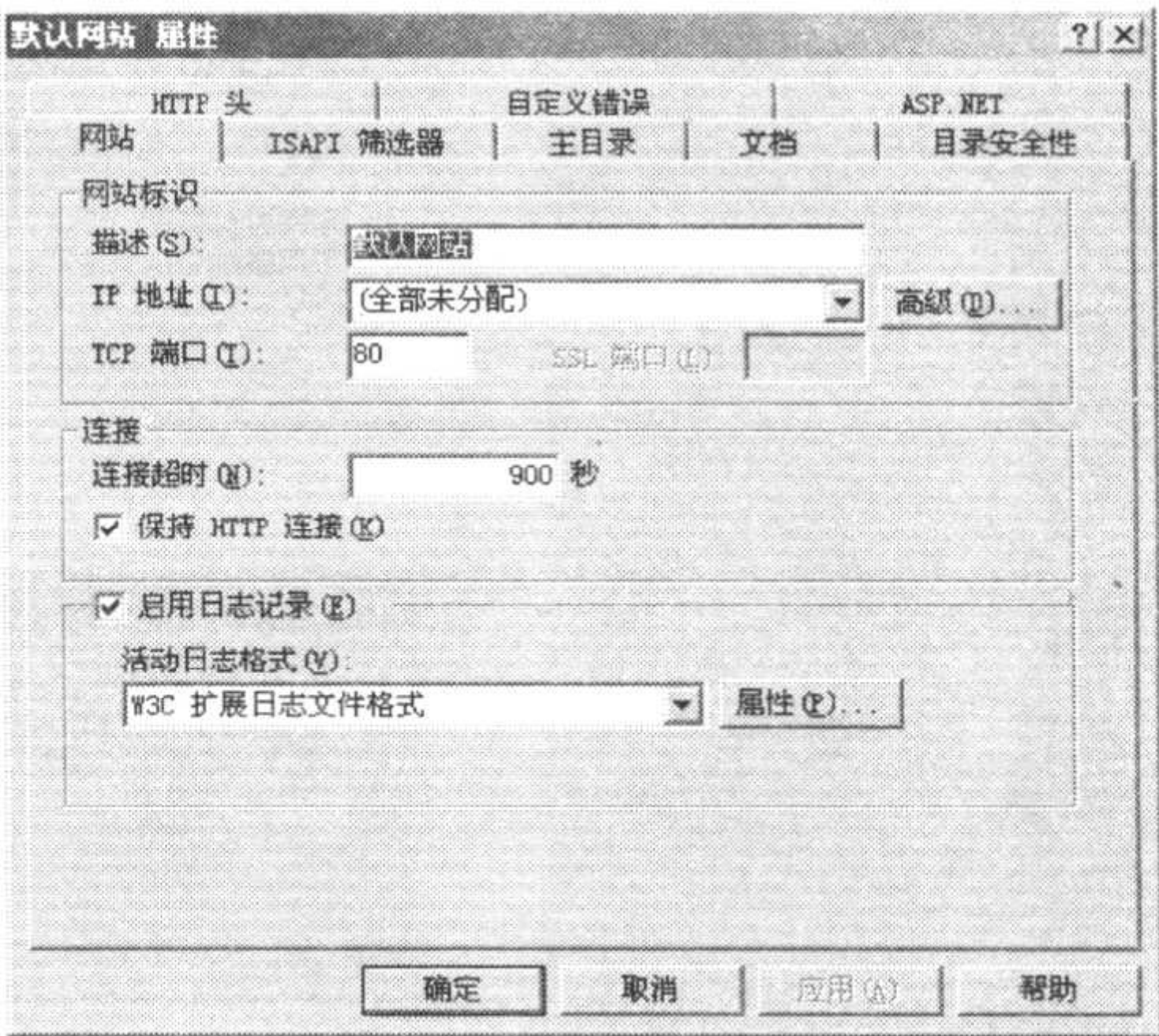


图 17-45 设置默认网站属性

(7) 单击【主目录】标签，单击【配置】按钮，出现如图 17-46 所示的对话框。

(8) 单击【添加】按钮，在弹出的对话框的【可执行文件】文本框中填入 Python 可执行文件的路径，并加上“%s %s”，例如“D:\Python25\python.exe %s %s”。在【扩展名】文本框中填入“.py”，如图 17-47 所示。单击【确定】按钮，使 IIS 支持 Python 脚本。



图 17-46 程序配置

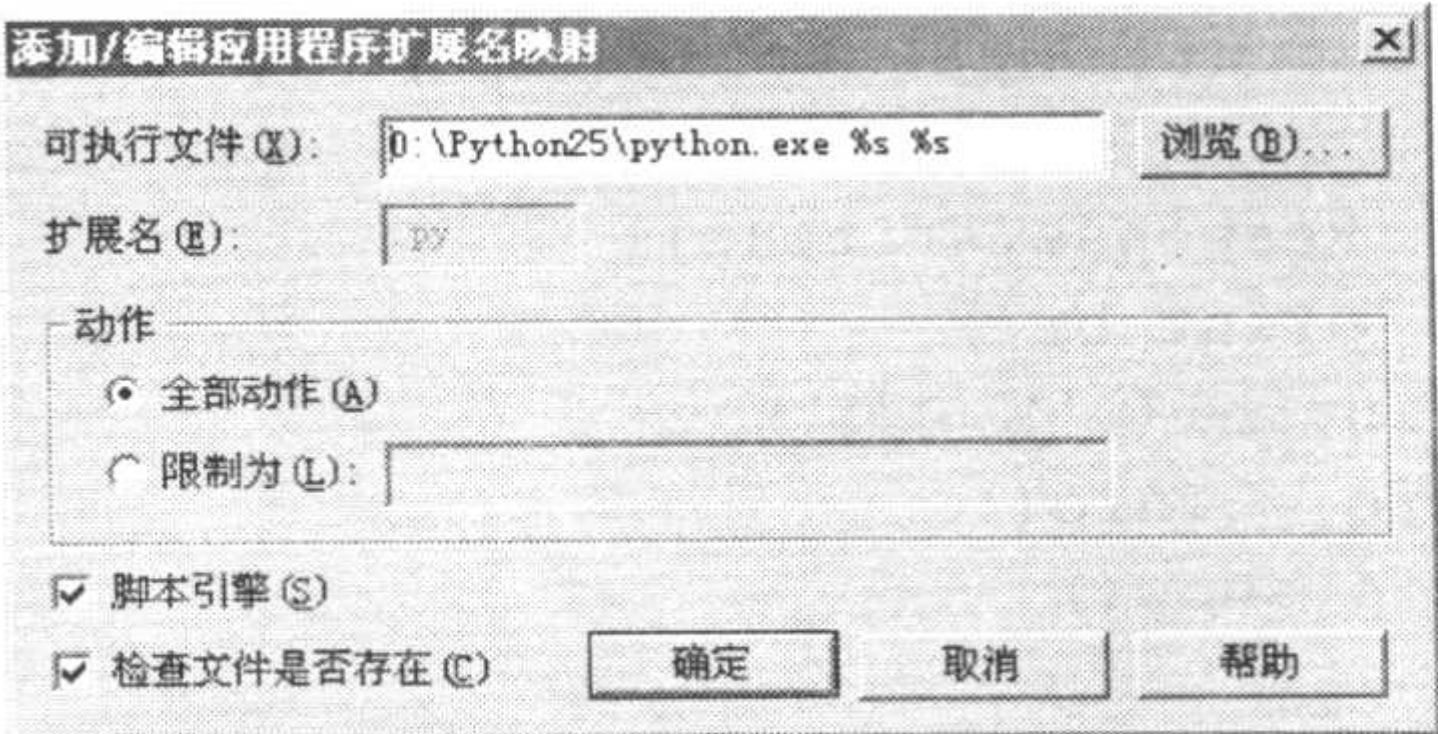


图 17-47 添加文件映射

17.3.2 在 ASP 中使用 Python 脚本

ASP 支持多种脚本语言。在“.asp”文件中可以像嵌入 VBscript 脚本一样嵌入 Python 脚本。也可以在 ASP 中直接使用 Python 脚本。在 ASP 中使用 Python 脚本基本上没有什么限制，与普通的脚本编写没有太大的区别。

1. 在“.asp”文件中包含 Python 脚本

在“.asp”文件中包含 Python 脚本时，需要在文件开头使用“<%@LANGUAGE=Python%>”指明。然后就可以在“<%”和“%>”之间使用 Python 脚本。需要注意的是，在“<%”和“%>”之间使用 Python 脚本仍要注意保持缩进。另外，Python 的 print 不能输出，需要使用 Response.Write 输入数字或者字符串。如下所示的 psptest.asp 在 ASP 中使用 Python。

```
<%@LANGUAGE=Python%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>use Python in ASP</title>
</head>
<body>
<h1>use Python in ASP</h1>
<%
import os                                # 导入 os 模块
import string                            # 导入 string 模块
class Info:                              # 定义类
    def _init_(self):
        Response.Write('<h1>Python Class </h1>')
    def show(self):
        Response.Write('<h1>Class Info</h1>')
def print_br():                           # 定义函数
    Response.Write('<br>')
def print_h1(s):
```



```

Response.Write('<h1>')
Response.Write(s)
Response.Write('</h1>')
print_h1(u'使用 os 模块')
for path in os.sys.path:
    Response.Write(path)
    print_br()
print_h1(u'使用 string 模块')
for s in string.split('Python is great'):
    Response.Write(s)
    print_br()
print_h1(u'使用类')
info = Info()
info.show()
%>
</body>
</html>

```

调用函数, 使用 u, 表示为 Unicode 编码
使用 os 模块
使用 string 模块
类实例化
调用类方法

脚本中使用 “<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />” 表示为 utf-8 编码, 在中文字符前使用 “u”, 表示为 Unicode 编码。psptest.asp 应保存为 utf-8 格式。将 psptest.asp 保存至 IIS 默认网站的目录 “C:\Inetpub\wwwroot”。在 IE 浏览器中输入 http://127.0.0.1/psptest.asp, 如图 17-48 所示。



图 17-48 在 IE 中打开 psptest.asp

2. 直接使用 Python 脚本

IIS 可以直接解释 Python 脚本。如果直接使用 Python 脚本代替 “.asp” 文件, 需要在 Python

脚本中输出 HTTP 状态代码。如下所示的 pythonasp.py 可以代替 “.asp” 文件。

```
# -*- coding:utf-8 -*-
# file: pythonasp.py
#
import os                                     # 导入 os 模块
def HttpStatus():                             # 定义函数输入 HTTP 状态
    print
    print 'Status: 200 OK'
    print 'Content-type: text/html'
    print
HttpStatus()                                  # 调用函数
print '''
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Python</title>
</head>
<body>
'''
print '<h1>Python 路径</h1>'
i = 1
for path in os.sys.path:                     # 使用 os 模块
    print i, ' ', path
    print '<br>'
    i = i + 1
print '''
</body>
</html>
'''
```

将 pythonasp.py 保存为 utf-8 格式, 将其保存至 IIS 默认网站的目录“C:\Inetpub\wwwroot”。在 IE 浏览器中输入 <http://127.0.0.1/pythonasp.py>, 如图 17-49 所示。



图 17-49 在 IE 中打开 pythonasp.py

17.3.3 一个简单的例子

从前边的例子可以看出，在 IIS 中使用 Python 和使用 Python 脚本没什么区别。一般地，在 ASP 中会使用 Access 作为数据库，而在使用了 Python 之后完全可以使用 SQLite 作为数据库。本小节中给出一个使用 SQLite 数据库的简单留言板的例子。首先在 SQLite 中创建一个存储留言的数据库，步骤如下所示（斜体部分为用户输入命令）。

```
E:\book\code>sqlite3.exe message
SQLite version 3.3.17
Enter ".help" for instructions
sqlite> CREATE TABLE message (name TEXT, mail TEXT, site TEXT, content TEXT, time
DATETIME);
sqlite> .exit
```

创建提交页面 submit.html，其内容如下所示。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>提交留言</title>
</head>

<body>
<table>
  <tr>
    <td>
      <h1>提交留言</h1>
      <br />
      <form id="form" name="form" method="post" action="addmessage.py">
        <label>姓名
        <input type="text" name="name" />
        </label>
        <p>
          <label>邮箱
          <input type="text" name="email" />
          </label>
        </p>
        <p>
          <label>网站
          <input type="text" name="site" />
          </label>
        </p>
        <p>
          <label>留言内容: <br />
          <textarea name="content" cols="50" rows="10"></textarea>
          </label>
        </p>
        <p>
```

```

        <input type="submit" name="Submit" value="提交" />
        <input name="Cancel" type="reset" id="Cancel" value="重置" />
    </p>
</form>
</td>
</tr>
</table>
</body>
</html>

```

创建添加留言页面 `addmessage.py` (应保存为 utf-8 格式), 其内容如下所示。

```

# -*- coding:utf-8 -*-
# file: addmessage.py
#
import cgi                                     # 导入 cgi 模块处理表单数据
import sqlite3                                 # 导入 sqlite3 模块
import datetime
print
print 'Status: 200 OK'
print 'Content-type: text/html'
print
form = cgi.FieldStorage()                     # 创建表单对象
name = unicode(form["name"].value, 'GBK')     # 获得表单变量, 并改变其编码
mail = unicode(form["email"].value, 'GBK')
site = unicode(form["site"].value, 'GBK')
content = unicode(form["content"].value, 'GBK')
now = datetime.datetime.now()                 # 获得当前时间
time = now.strftime('%Y-%m-%d %H:%M:%S')      # 格式化时间
con = sqlite3.connect('message')              # 连接到数据库
cur = con.cursor()
cur.execute("INSERT INTO message VALUES(?,?,?,?,?)", (name, mail, site, content, time))
con.commit()
cur.close()
con.close()
print '''
<html>
<head>
<title>添加成功</title>
</head>
<body>
<h1>添加成功</h1>
<br>
<a href=show.py>单击查看留言</a>
</body>
</html>
'''

```

创建查看留言页面 `show.py` (应保存为 utf-8 格式), 其内容如下所示。

```

# -*- coding:utf-8 -*-

```



```
# file: PySqlite.py
#
import sqlite3                                     # 导入 sqlite3 模块
con = sqlite3.connect('message')                  # 连接到数据库
cur = con.cursor()                                # 获得数据库游标
cur.execute('select * from message')              # 执行 SQL 语句
results = cur.fetchall()                          # 获得数据
print
print 'Status: 200 OK'
print 'Content-type: text/html'
print
print '''
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>use Python in ASP</title>
</head>
<body>
<center>
<h1>所有留言</h1>
</center>
<hr />
'''
for result in results:
    print '姓名:', result[0].encode('UTF-8')
    print '<br>'
    print '时间:', result[4].encode('UTF-8')
    print '<br>'
    print '邮箱:', result[1].encode('UTF-8')
    print '<br>'
    print '网站', result[2].encode('UTF-8')
    print '<br>'
    print '留言内容:'
    print '<br>'
    print result[3].encode('UTF-8')
    print '<hr />'
print '''
</body>
</html>
'''
cur.close()                                       # 关闭游标
con.close()                                     # 关闭数据库连接
```

将上述的页面和 message 数据库保存到 IIS 的网站目录，默认为 “C:\Inetpub\wwwroot”，在 IE 地址栏中输入 <http://127.0.0.1/submit.html>，然后输入留言内容，如图 17-50 所示。单击【提交】按钮，将打开添加留言页面，单击“查看留言”连接，如图 17-51 所示。该留言板很简单，只作演示用，读者可以自己将其完善。

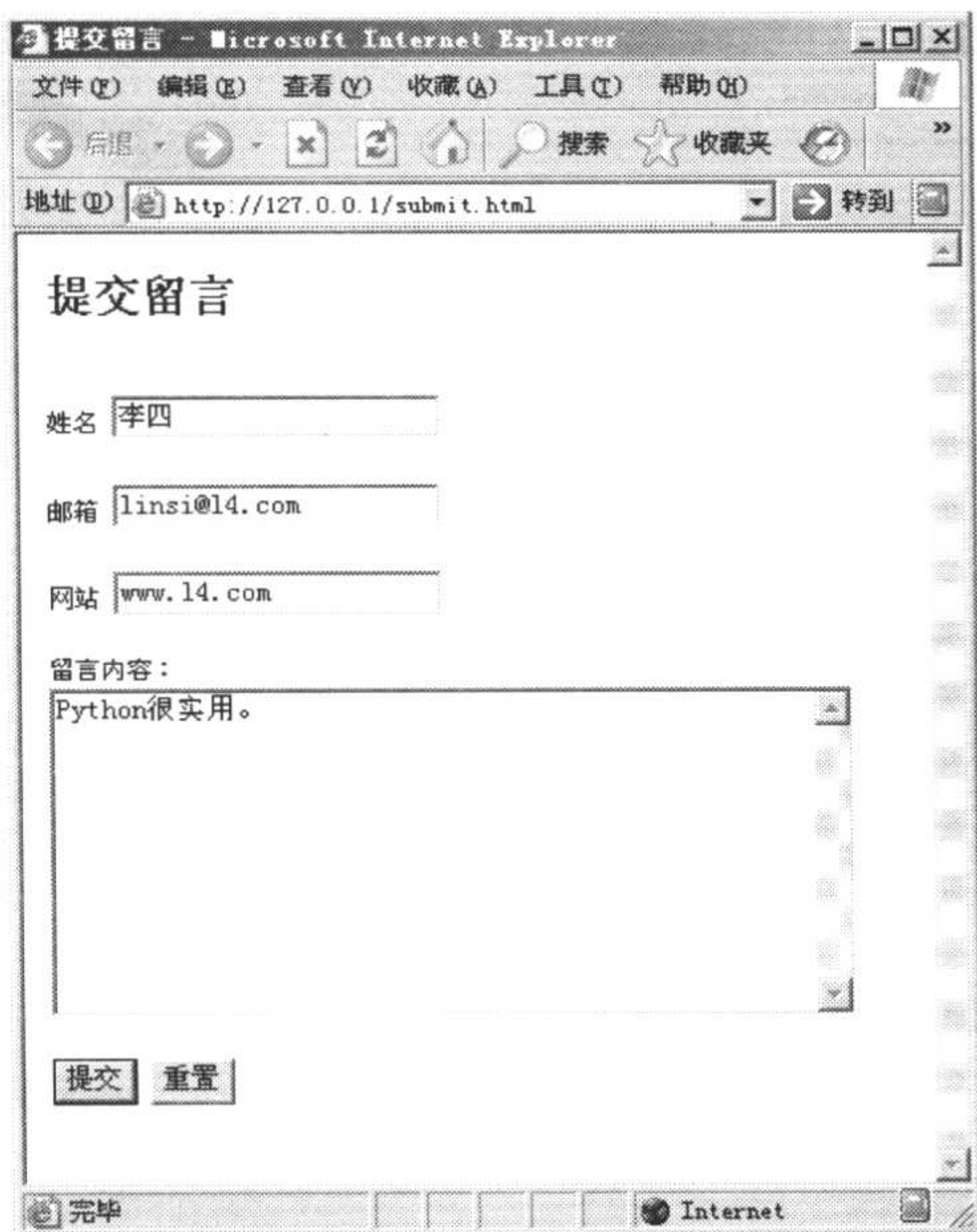


图 17-50 提交留言



图 17-51 查看留言

17.4 在 Apache 中使用 Python

Apache 是安全性很高、非常流行的开源 HTTP 服务器，可以运行在多种操作系统中。在 Apache 中可以使用 Perl、Python、PHP 等作为编程语言。在 Apache 中使用 Python 需要下载安装 mod_python 模块。

17.4.1 安装配置 Apache

Apache 提供了 Windows 下的安装程序，使用 Apache 的安装程序可以方便地完成 Apache 的安装。由于 Apache 使用文本文件作为配置文件，其配置过程较为繁琐。

1. 安装 Apache

Apache 的安装程序可以从其官方网站 <http://www.apache.org> 下载，以 apache_2.2.4-win32-x86-openssl-0.9.8d.msi 为例，其安装步骤如下所示。

- (1) 双击运行安装程序，如图 17-52 所示。
- (2) 单击【Next】按钮，进入安装协议界面，如图 17-53 所示。
- (3) 单击【Next】按钮，进入安装说明界面，如图 17-54 所示。
- (4) 单击【Next】按钮，进入服务器信息设置界面，此处可以设置服务器的域名、主机名等。如果没有域名，可以不填写，如图 17-55 所示。
- (5) 单击【Next】按钮，进入安装类型选择界面，如图 17-56 所示。



图 17-52 Apache 安装界面

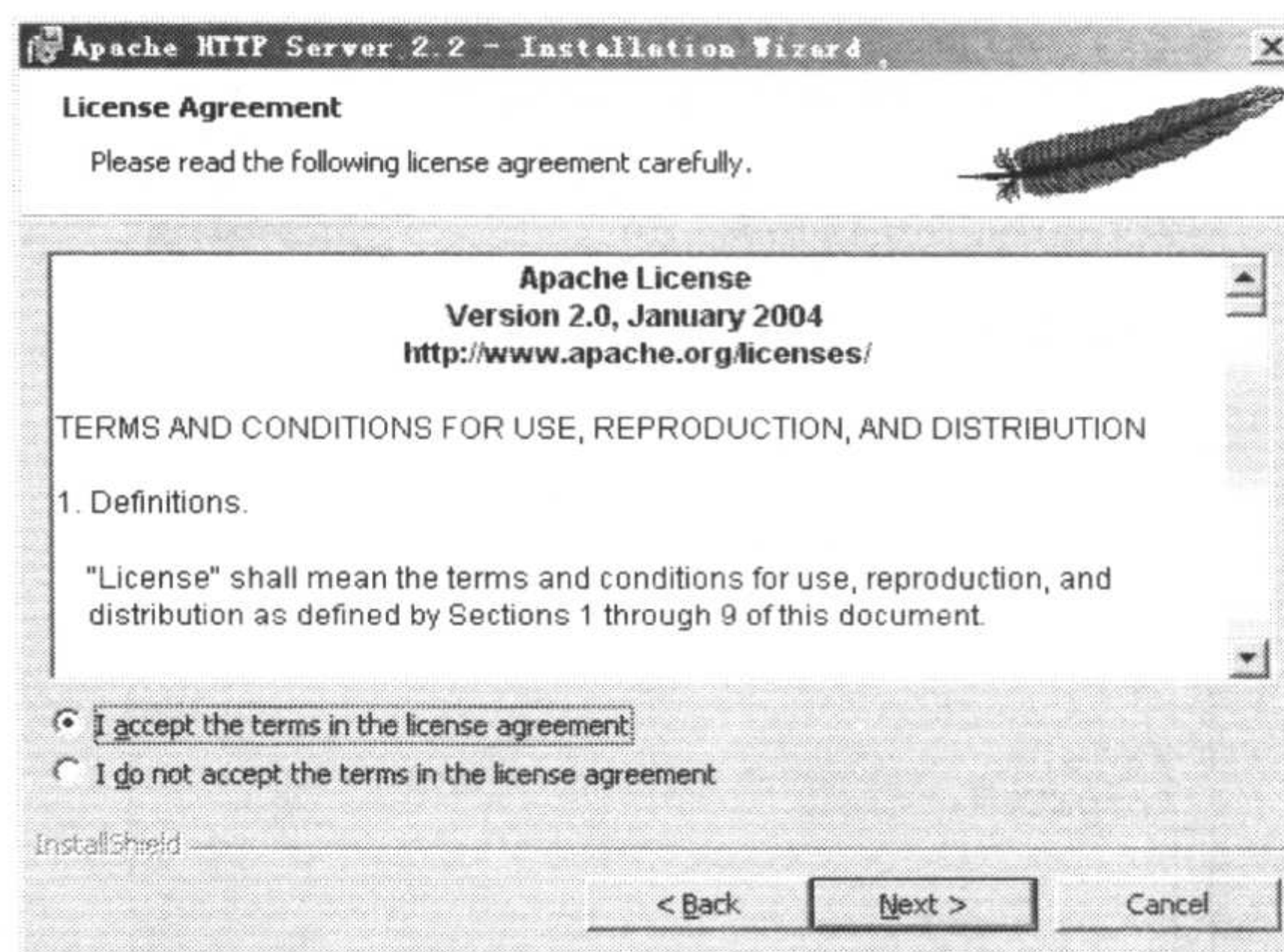


图 17-53 Apache 安装协议

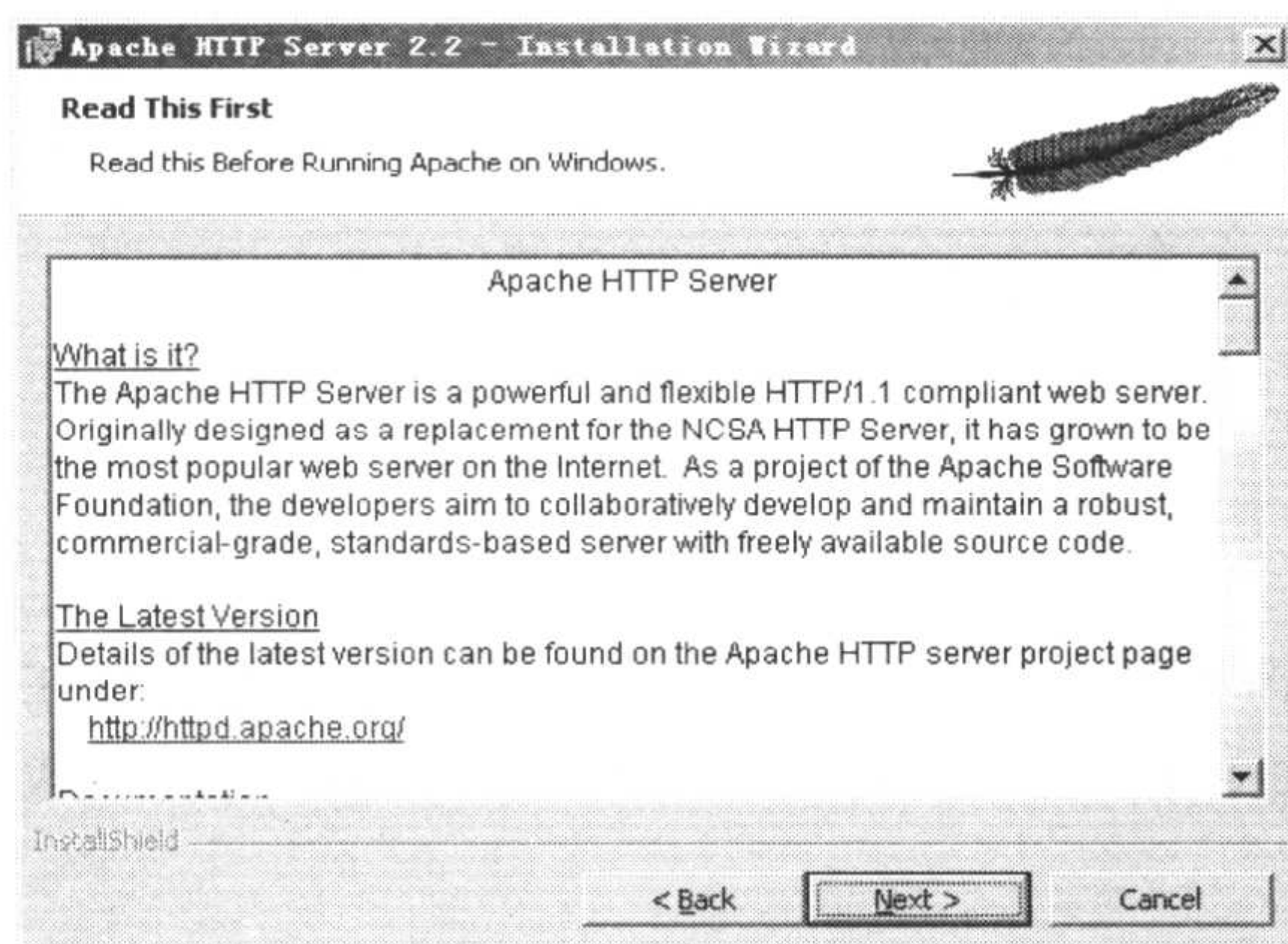


图 17-54 Apache 安装说明

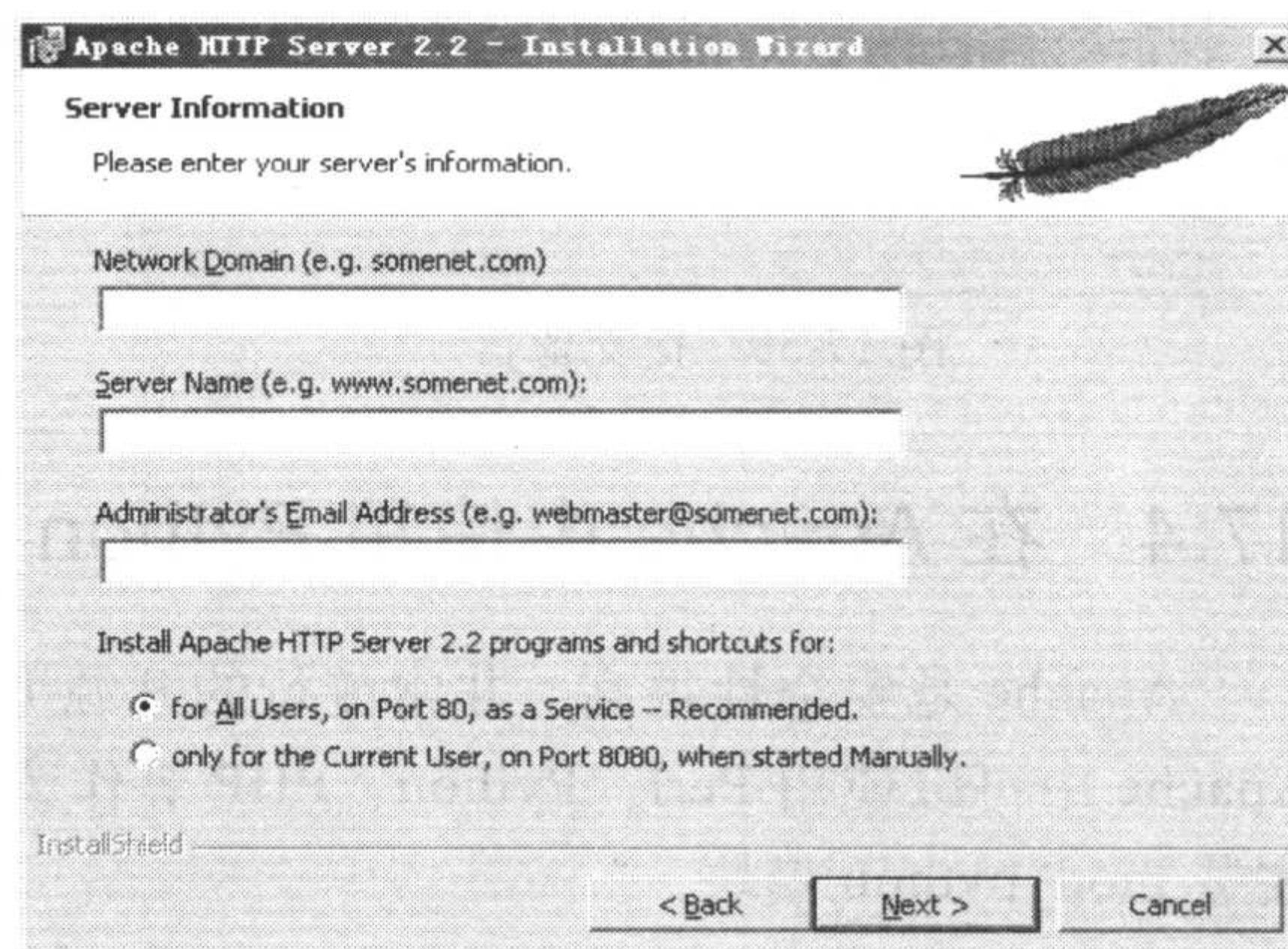


图 17-55 服务器信息设置

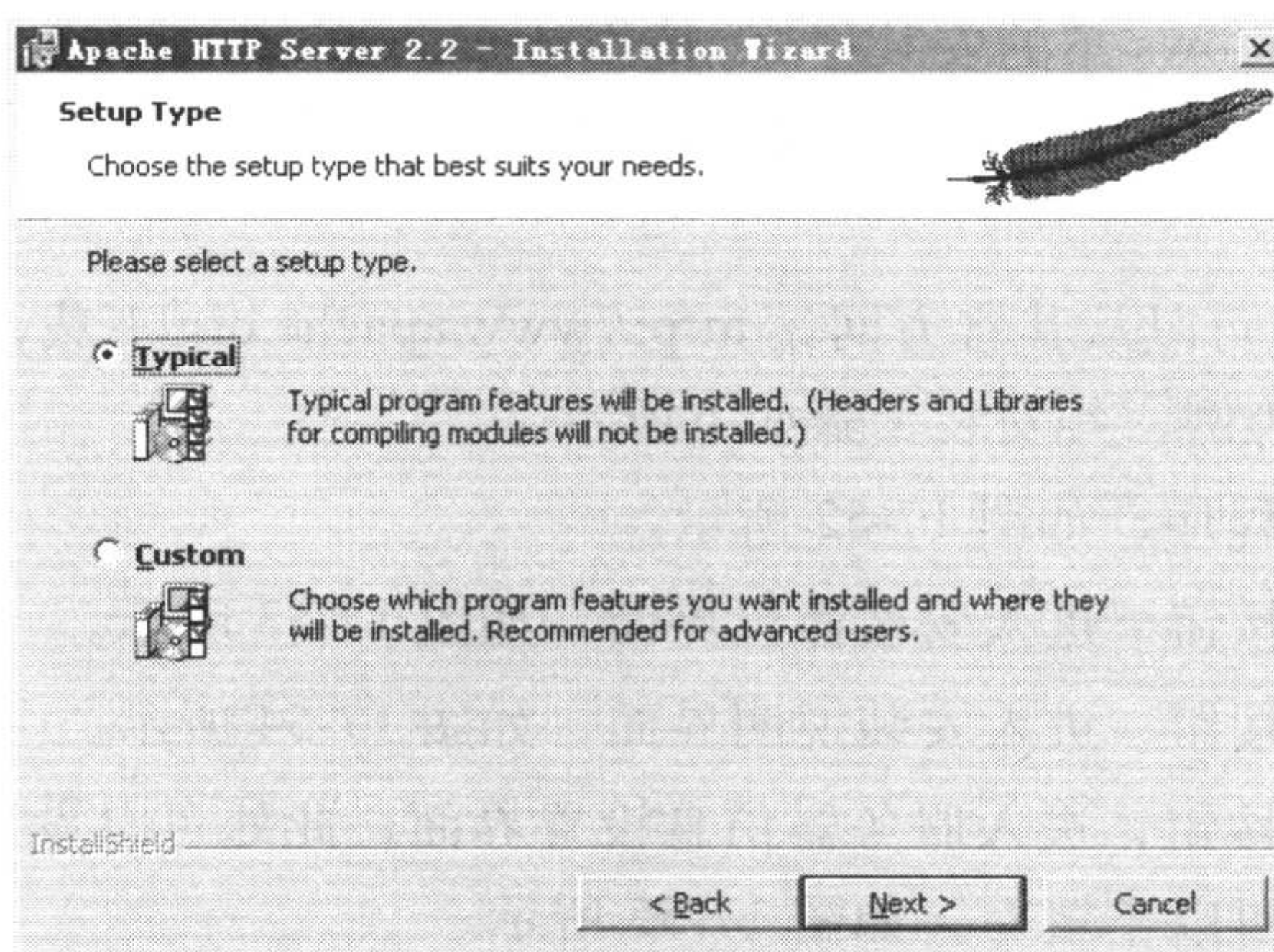


图 17-56 安装类型选择

(6) 单击【Next】按钮，进入安装路径选择界面，此处可以根据需要设置 Apache 的安装路径，如图 17-57 所示。

(7) 单击【Next】按钮，进入安装确认界面，如图 17-58 所示，单击【Install】按钮安装 Apache。

(8) 安装完成后，安装程序会将 Apache 设为 Windows 服务，并启动 Apache。

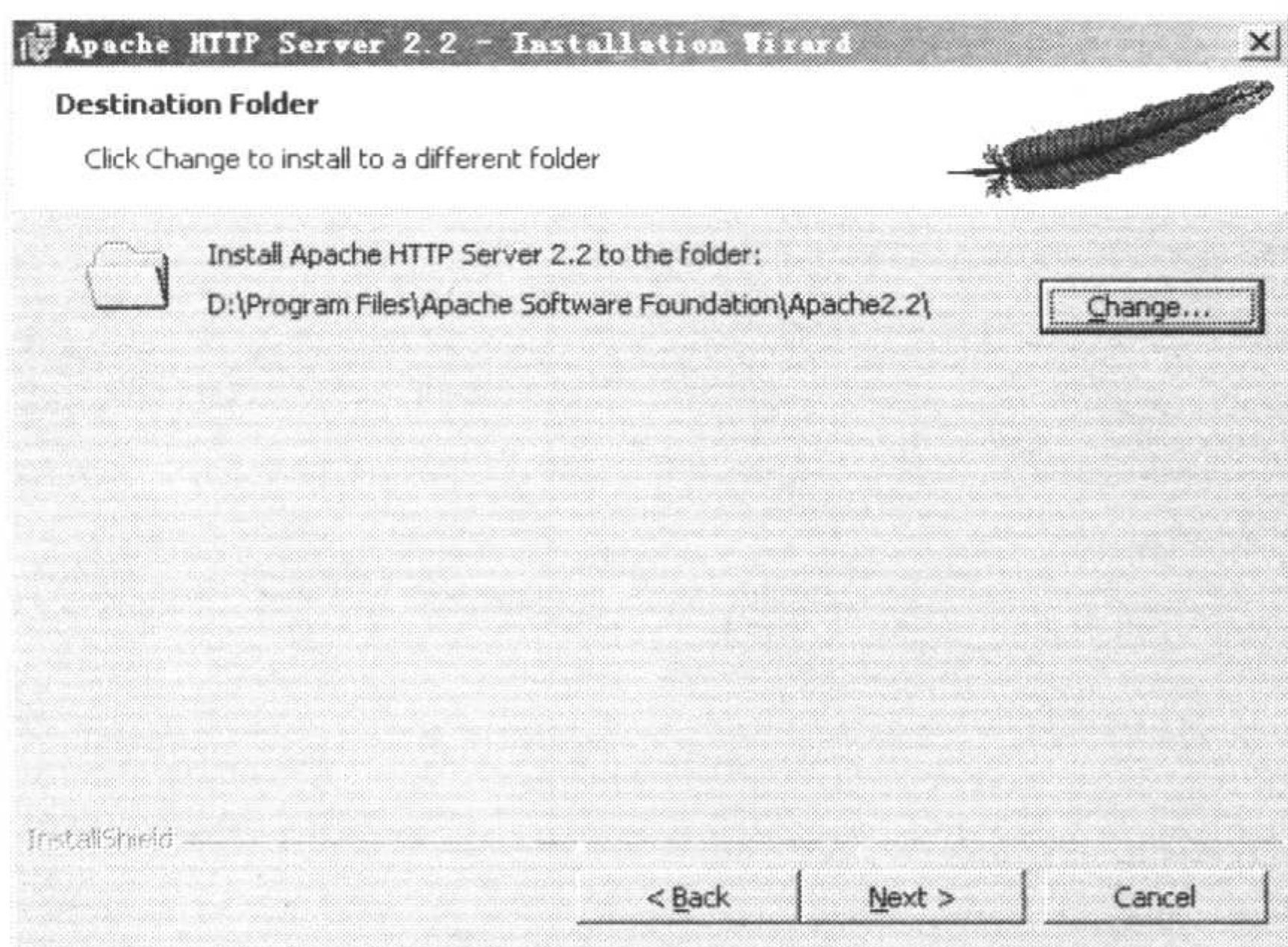


图 17-57 安装路径选择

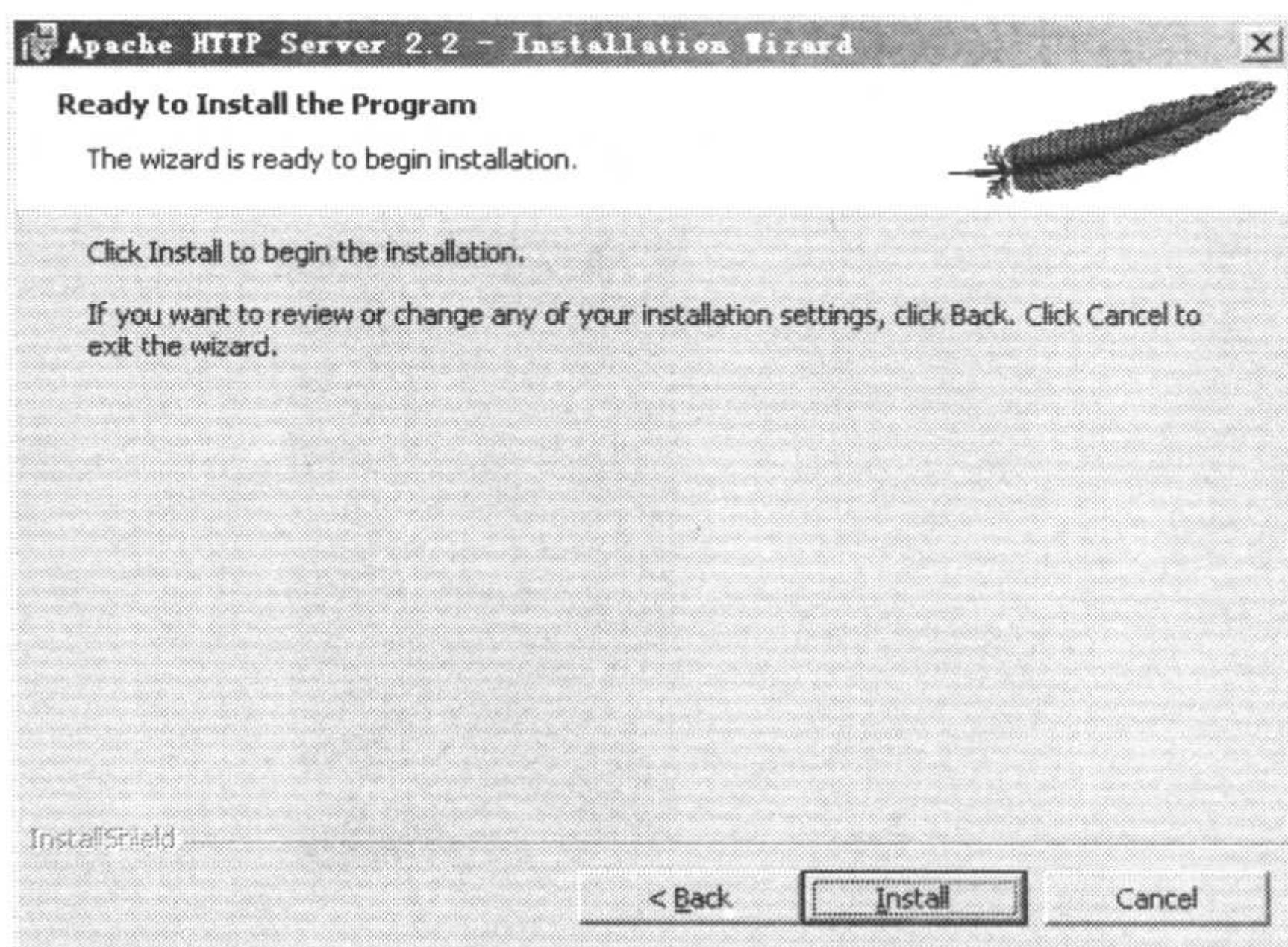


图 17-58 确认安装

2. 配置 Apache

如果已经安装了 Microsoft IIS，并且在安装过程中没有使用正确的域名，Apache 可能会出现错误，此时需要对 Apache 进行配置，Apache 的配置过程如下所示。

(1) 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Configure Apache Server】|【Edit the Apache httpd.conf Configuration File】命令，将打开 Apache 配置文件。

(2) 将第 133 行的“ServerAdmin”使用“#”注释掉。

(3) 将第 142 行的“ServerName :80”使用“#”注释掉。

(4) 如果安装了 Microsoft IIS，则需要修改 Apache 监听的端口，将第 23 行的“Listen 80”修改为“Listen 82”。

(5) 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Control Apache Server】|【Restart】命令，重新启动 Apache。

(6) 在 IE 地址栏中输入 `http://127.0.0.1:82`，或者 `http://localhost:82`，将显示“It works!”，表明 Apache 已经正常工作。

17.4.2 安装 mod_python

mod_python 是用于使 Apache 支持 Python 脚本的模块，使用它可以在 Apache 中使用 PSP (Python Server Pages) 或者使用 Python 编写 CGI。mod_python 可以从其官方网站

<http://www.modpython.org> 下载。

mod_python 不仅作为 Apache 的模块，也作为 Python 的模块被安装到 Python 中，因此需要根据所安装的 Python 的版本，从其官方网站下载相应的 mod_python 安装文件。以 Python 2.5 为例，mod_python 的安装配置过程如下所示。

(1) 从 mod_python 官方网站下载 mod_python-3.3.1.win32-py2.5-Apache2.2.exe 安装程序，双击运行安装程序，如图 17-59 所示。

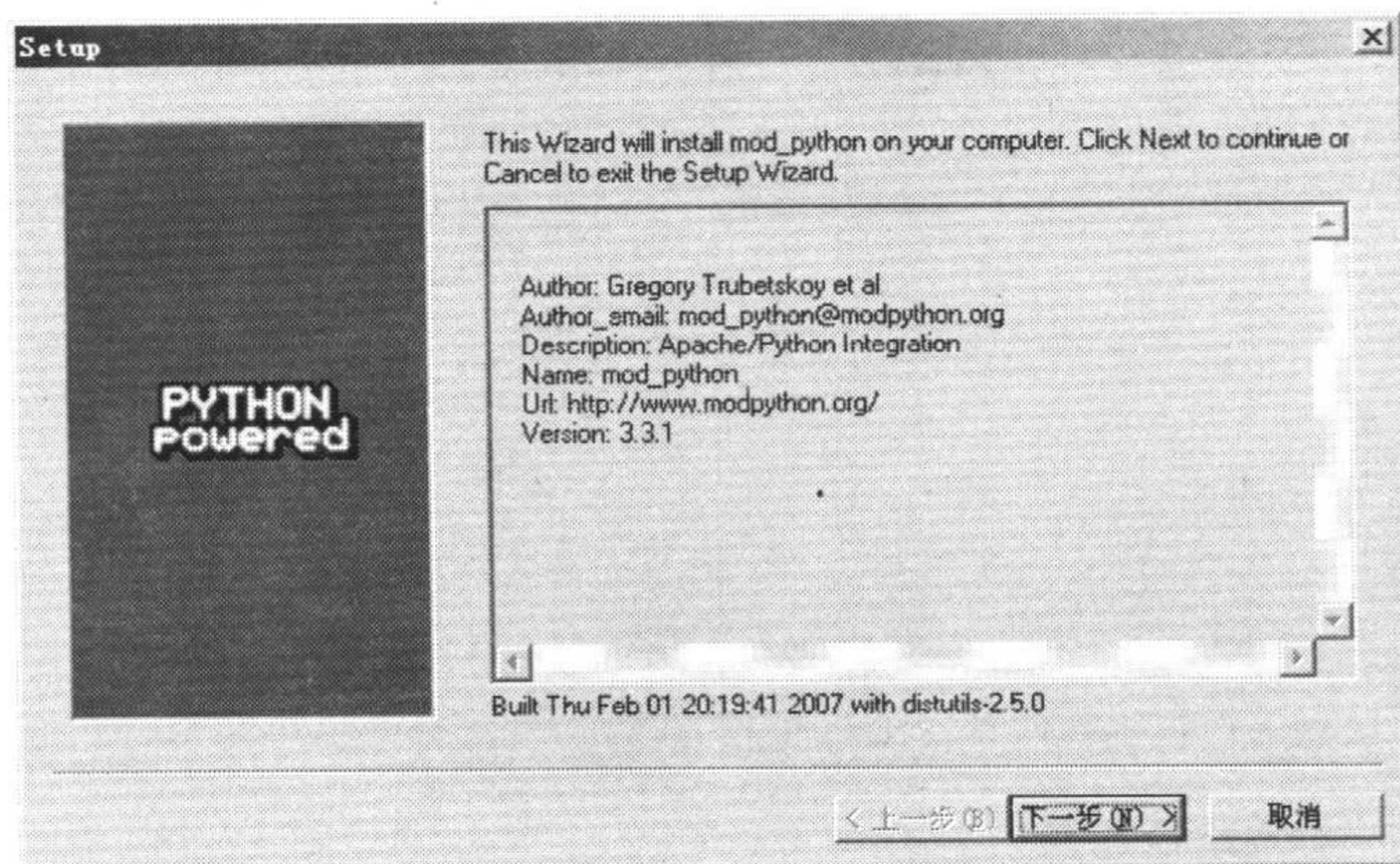


图 17-59 安装 mod_python

(2) 单击【下一步】按钮，进入安装路径界面，如图 17-60 所示。

(3) 单击【下一步】按钮进行安装，在 mod_python 的安装过程中将会弹出如图 17-61 所示的设置 Apache 安装路径的界面。

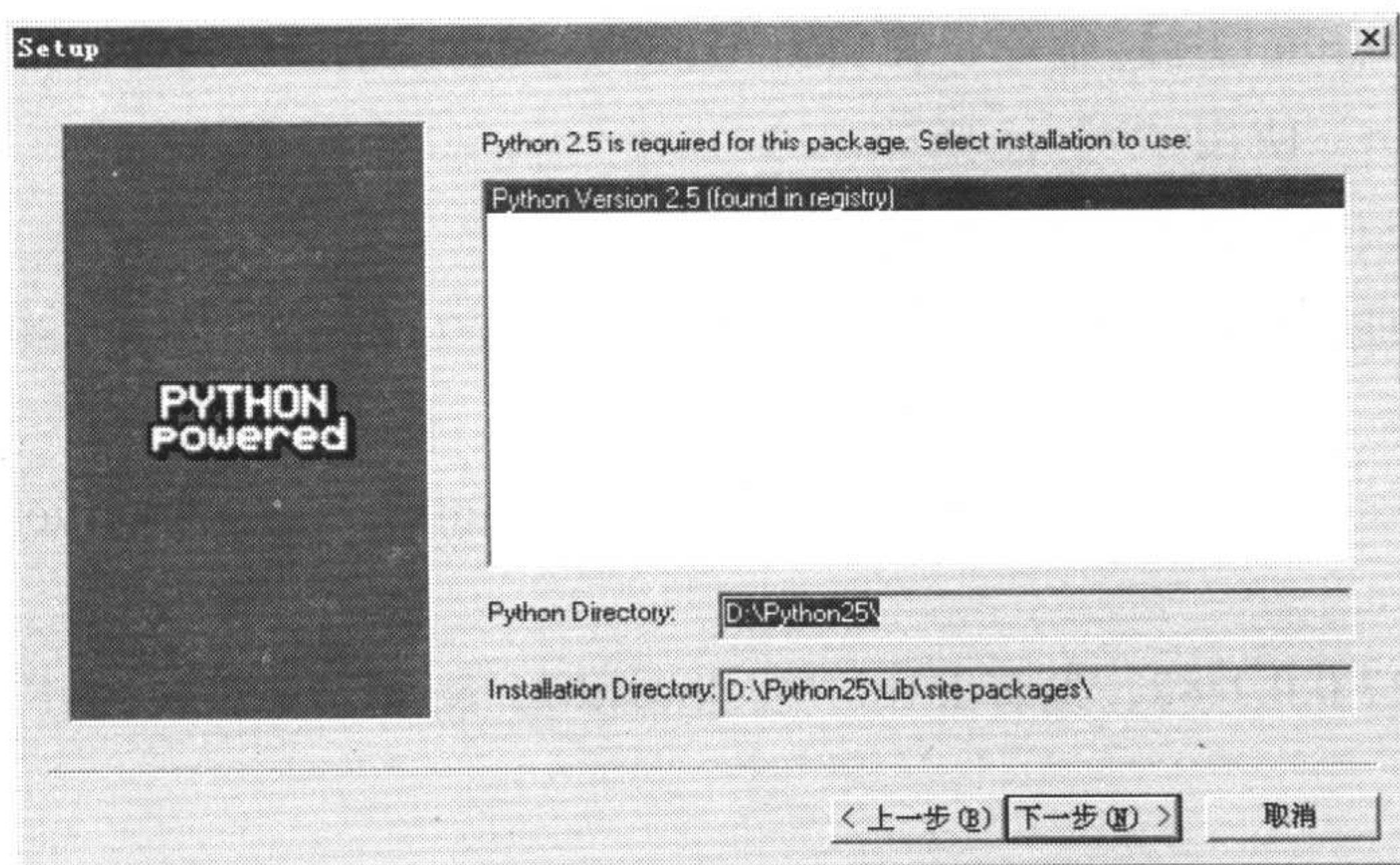


图 17-60 mod_python 安装路径

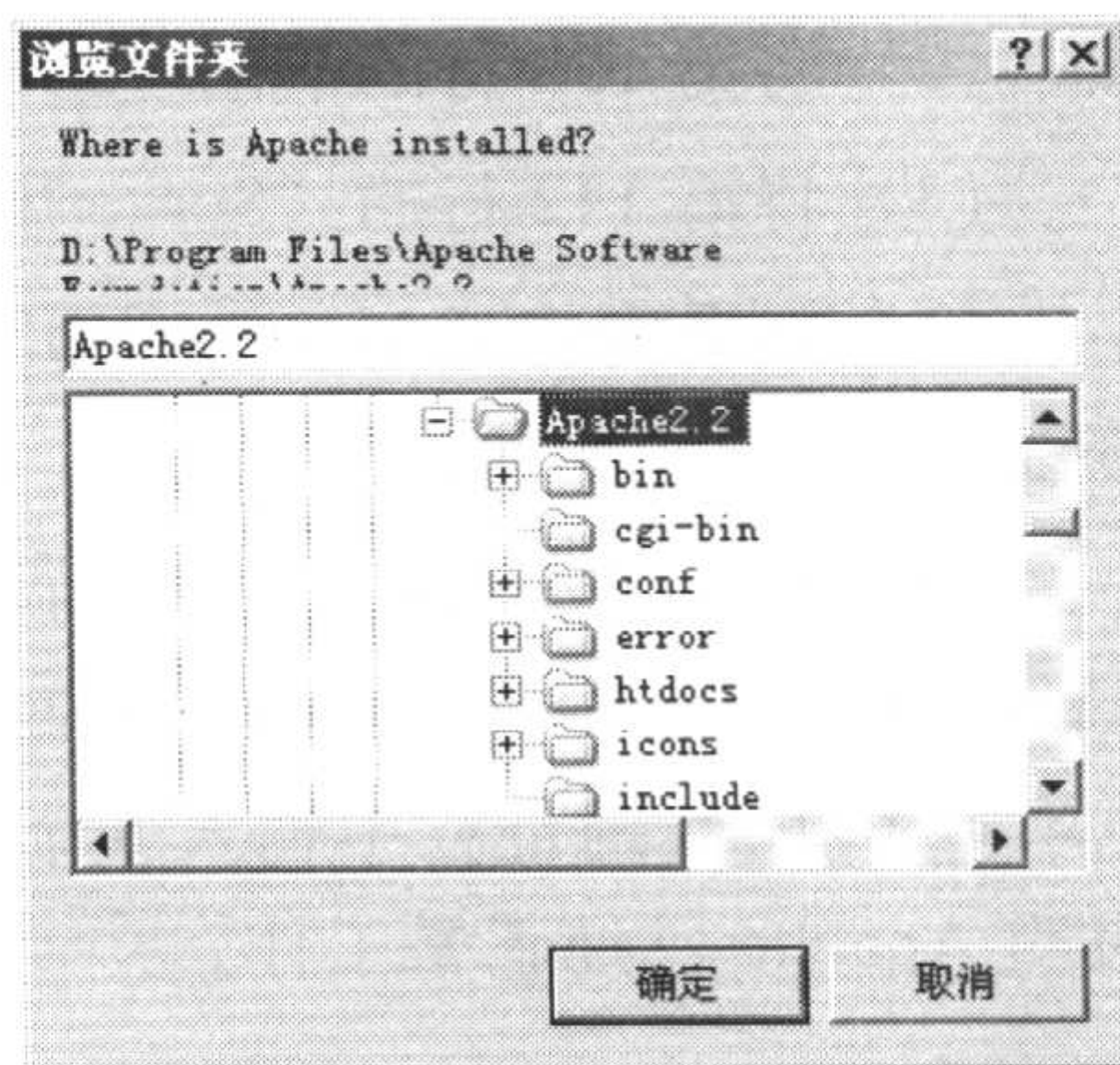


图 17-61 选择 Apache 安装路径

(4) 安装完成后，单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Configure Apache Server】|【Edit the Apache httpd.conf Configuration File】命令，打开 Apache 配置文件，

在第 66 行以后添加如下所示语句，在 Apache 中载入 mod_python 模块。

```
LoadModule python_module modules/mod_python.so
```

(5) 在<Directory "D:/Program Files/Apache Software Foundation/Apache2.2/htdocs">（根据 Apache 所安装的路径，此处可能会有不同）和</Directory>之间添加如下所示的语句。

```
AddHandler mod_python .py
PythonHandler pythontest
PythonDebug On
```

(6) 在"D:/Program Files/Apache Software Foundation/Apache2.2/htdocs"编写 pythontest.py，其内容如下所示。

```
# -*- coding:utf-8 -*-
# file: pythontest.py
#
from mod_python import apache
def handler(req):
    req.content_type = 'text/html'
    req.write('''
<html>
<head>
<title>Python</title>
</head>
<body>
<h1>mod_python</h1>
</body>
</html>
''')
    return apache.OK
```

(7) 单击【开始】|【所有程序】|【Apache HTTP Server 2.2.4】|【Control Apache Server】|【Restart】命令，重新启动 Apache。

(8) 在 IE 地址栏中输入 http://127.0.0.1:82/pythontest.py 或者 http://localhost:82/pythontest.py，将显示“mod_python”，表明 mod_python 可以使用了。

17.4.3 使用 Python Sever Pages 创建留言板

在上一节测试的例子中在 Apache 配置文件中添加“PythonHandler pythontest”语句，并添加了一个发布处理器。发布处理器是一个 Python 脚本，mod_python 使用该脚本来处理 Apache 发出的请求。配置文件中“AddHandler mod_python .py”表示所有的“.py”文件都由 mod_python 处理。如果觉得自己创建发布处理器比较麻烦，可以使用 mod_python 的标准发布处理器。将“PythonHandler pythontest”修改为“PythonHandler mod_python.publisher”即可。

mod_python 还支持 PSP (Python Sever Pages)，可以使用类似于 ASP 的语法嵌入 Python。为了使 Apache 支持 PSP，需要将“AddHandler mod_python .py”修改为“AddHandler

mod_python.psp”，将“PythonHandler pythontest”修改为“PythonHandler mod_python.psp”。重新启动 Apache 后就可以在 Apache 中使用 PSP 创建页面。

在本小节中将修改上一节中的使用 SQLite 制作留言板的例子，将数据库修改为 MySQL，环境改为 Apache。为了让 MySQL 支持中文，首先应修改其配置文件 my.ini。my.ini 位于 MySQL 的安装目录中，例如“D:\Program Files\MySQL\MySQL Server 5.0”。将“default-character-set”项值该为“gbk”，如下所示。

```
[mysql]
default-character-set=gbk
```

修改后重新启动 MySQL 服务。单击【开始】|【所有程序】|【MySQL】|【MySQL Server 5.0】|【MySQL Command Line Client】命令，输入如下命令创建留言本的数据库（斜体部分为用户输入命令）。

```
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 17
Server version: 5.0.37-community-nt MySQL Community Edition (GPL)
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>CREATE DATABASE message DEFAULT CHARACTER SET gbk COLLATE gbk_chinese_ci;
Query OK, 1 row affected (0.02 sec)
mysql> USE message;
Database changed
mysql> CREATE TABLE message(name TEXT, mail TEXT, site TEXT, content TEXT, time
DATETIME);
Query OK, 0 rows affected (0.13 sec)
mysql> exit
```

将上一节中的 submit.html 中的如下语句。

```
<form id="form" name="form" method="post" action="addmessage.py">
```

修改为如下所示。

```
<form id="form" name="form" method="post" action="addmessage.psp">
```

编写如下所示的添加留言页面 addmessage.psp。

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>添加成功</title>
</head>
<body>
<%
import MySQLdb                                # 导入 MySQLdb 模块
import datetime
name = req.form['name']                        # 获取表单内容
mail = req.form['email']
site = req.form['site']
content = req.form['content']
now = datetime.datetime.now()
```

```

time = now.strftime('%Y-%m-%d %H:%M:%S')
db = MySQLdb.connect(host='localhost',
                      user='root',
                      passwd='python',
                      db='message',
                      charset='gb2312')
cur = db.cursor()
cur.execute("INSERT INTO message VALUES('%s','%s','%s','%s','%s')" % (name,mail,site,
content,time))
db.commit()
cur.close()
db.close()
%>
<h1>添加成功</h1>
<br>
<a href=show.psp>单击查看留言</a>
</body>
</html>

```

编写如下所示的查看留言页面 show.psp。

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>use Python in ASP</title>
</head>
<body>
<center>
<h1>所有留言</h1>
</center>
<hr />
<%
import MySQLdb
db = MySQLdb.connect(host='localhost',
                      user='root',
                      passwd='python',
                      db='message',
                      charset='gb2312')
cur = db.cursor()
cur.execute('select * from message')
results = cur.fetchall()
for result in results:
    req.write( '姓名: %s' % result[0].encode('gb2312') )
    req.write( '<br>' )
    req.write( '时间: %s' % result[4] )
    req.write( '<br>' )
    req.write( '邮箱: %s' % result[1].encode('gb2312') )
    req.write( '<br>' )
    req.write( '网站: %s' % result[2].encode('gb2312') )
    req.write( '<br>' )
    req.write( '留言内容:' )

```



```
req.write( '<br>' )
req.write( result[3].encode('gb2312') )
req.write( '<hr />' )
cur.close()
db.close()
%>
</body>
</html>
```

将上述的页面保存到 Apache 网站发布目录，如 “D:\Program Files\Apache Software Foundation\Apache2.2\htdocs” 中，在 IE 地址栏中输入 <http://127.0.0.1:82/submit.html>，就可以提交、查看留言。

第 18 章 Python 网络编程

Python 的标准模块中提供了对网络编程的支持。在 Python 中，可以使用 socket 模块进行底层的网络编程，也可以使用 urllib、httplib、ftplib、poplib 和 smtpplib 等模块针对特定的网络协议进行编程。除了 Python 的标准模块以外，还可以使用 Twisted 进行网络编程。Twisted 支持多种底层协议，使用 Twisted 可以更加方便地编写网络应用程序。

18.1 使用 socket 模块

Python 中的 socket 模块提供了底层的网络接口，使用 Python 的 socket 模块可以实现网络上不同计算机之间的 socket 通信。Python 中的 socket 实现了 BSD (Berkeley Software Distribution) 套接字标准。

18.1.1 网络编程概述

网络服务都是建立在 socket 基础之上。socket 是网络连接端点，是网络的基础。每个 socket 都被绑定到指定的 IP 和端口上。例如，上一章中使用 127.0.0.1:8080 来访问 Zope 构建的网站。其中 127.0.0.1 是一个特殊的 IP 地址，它总是指向本机。而 IP 地址后的 8080 则是 Zope 服务所监听的端口。

每台处于网络中的计算机都有一个 IP 地址来标识自己的位置。IP 地址是一个 32 位长的二进制数，为方便起见，IP 地址通常由小于 255 的 4 组整数组成，每组数之间用“.”隔开。例如“123.45.67.89”，如果将其写成二进制的形式，则是“01111011001011010100001101011001”。有些 IP 地址比较特别，例如 127.0.0.1，它总是指向本机，而以“192.168.*.*”开头的则只能用于局域网中。

IP 地址还可以和域名绑定，例如，IP 地址 82.94.237.218 绑定到了 www.python.org 域名上。使用 82.94.237.218 同样可以访问 Python 的官方网站，但显然 IP 地址不如域名更容易记忆。

除了 IP 地址，使用 socket 时还需要指定计算机端口。计算机端口的取值范围是 0~65535，其中小于 1024 的都是系统所保留的端口，或者一些网络服务所使用的端口。例如，FTP 服务使用 21 端口，Web 服务使用 80 端口，系统也使用一些较大的端口，例如，Windows 系统中

著名的 3389 端口。

IP 和端口表明了计算机在网络中的位置，类似于邮政编码和详细地址。计算机之间为了进行通信还需要遵循特定的计算机网络协议。常见的网络协议有以下几种。

- TCP/IP 协议，即传输控制协议/互联网协议，其用于在安装了不同硬件和不同操作系统的计算机之间实现可靠的网络通信。其中，TCP 协议用于保证数据包传输的可靠性；IP 协议用于保证数据包能被传送到目标计算机。
- NetBIOS 协议，其是由 IBM 公司开发的，主要用于小型局域网。NetBIOS 协议为程序提供了请求低级服务的统一的命令接口，几乎所有的局域网都是在 NetBIOS 协议的基础上工作的。
- FTP 协议，即文件传输协议。FTP 是 Internet 中使用得最多的文件传输协议。通过 FTP 协议可以使用客户端从 FTP 服务器上下载的各种文件。
- Telnet 协议，即远程登录协议，使用 Telnet 协议可以登录到远程计算机。目前仍有很多的 BBS 可以使用 Telnet 登录。
- HTTP 协议，即超文本传输协议，其用于传送 WWW 方式的数据。上一章中所创建的 Web 应用都是基于 HTTP 协议的。
- PPP 协议，即点对点协议，其主要用来创建电话线路以及 ISDN 拨号接入 ISP 的连接，具有多种身份验证方法、数据压缩和加密以及通知 IP 地址等功能。
- PPPoE 协议，即以太网上的点对点协议，其广泛用于 ADSL 接入方式，也就是常说的宽带。通过 PPPoE 技术和宽带调制解调器，用户可以创建虚拟拨号连接，连接到 Internet 上。

18.1.2 使用 socket 模块建立网络通信

使用 Python 中 socket 模块提供的 socket 对象的方法，可以在计算机之间建立连接。一般来说，使用 socket 创建的通信应有服务端和客户端，服务端首先建立一个 socket，并等待客户端的连接。客户端建立与服务端的 socket 连接，当连接成功后，客户端和服务端就可以使用 socket 进行通信。

1. socket 模块简介

使用 socket 模块时，应首先使用 socket() 函数，创建一个 socket 对象。然后就可以使用 socket 对象的方法创建连接。socket() 函数的原型如下所示。

```
socket( family, type, proto)
```

其参数含义如下。

- family: 地址系列，可选参数。默认为 AF_INET，也可以是 AF_INET6 或 AF_UNIX。
- type: socket 类型，可选参数。默认为 SOCK_STREAM。
- proto: 协议类型，可选参数。

创建好 socket 对象后，可以使用 socket 对象的 bind 方法绑定 IP 地址和端口。bind 方法

的原型如下所示。

```
bind(address)
```

其参数含义如下。

- **address**: 由 IP 地址和端口组成的元组, 例如 “(‘127.0.0.1’,1051)”。如果 IP 地址为空, 则表示本机。

使用 socket 对象的 `listen` 方法可以监听所有 socket 对象创建的连接。其函数原型如下所示。

```
listen(backlog)
```

其参数含义如下。

- **backlog**: 指定连接队列数, 最小值为 1, 最大值由所使用的操作系统决定, 一般情况下为 5。

使用 socket 对象的 `connect` 和 `connect_ex` 都可以连接到服务端, 不同的是将返回一个错误, 代替引发一个异常。其函数原型分别如下所示。

```
connect(address)
```

```
connect_ex(address)
```

其参数含义如下。

- **address**: 由 IP 地址和端口组成的元组。

使用 socket 对象的 `accept` 方法可以接收来自客户端的数据, `accept` 方法将返回一个新的 socket 对象和客户端的地址。使用 socket 对象的 `recv` 和 `recvfrom` 方法都可以从 socket 对象获取数据, 不同的是 `recvfrom` 方法返回所接收的字符串和地址, 而 `recv` 方法仅返回字符串, 其原型分别如下所示。

```
recv(bufsize, flags)
```

```
recvfrom(bufsize, flags)
```

其参数含义如下。

- **bufsize**: 指定接收缓冲区的大小。
- **flags**: 接收标志, 可选参数。

使用 socket 对象的 `send` 和 `sendall` 方法都可以向已经连接的 socket 发送数据, 不同的是 `sendall` 将一直发送完全部数据, 其原型分别如下所示。

```
send(string, flags)
```

```
sendall(string, flags)
```

其参数含义如下。

- **string**: 所发送的数据。
- **flags**: 发送标志, 可选参数。

使用 socket 对象的 `sendto` 方法可以向一个未连接的 socket 发送数据, 其参数原型如下所示。

```
sendto(string, flags, address)
```

其参数含义如下。

- string: 所发送的数据。
- flags: 发送标志, 可选参数。
- address: 由 IP 地址和端口组成的元组。

使用 socket 对象的 makefile 方法可以将 socket 关联到文件对象上, 其原型如下所示。

```
makefile(mode, bufsize)
```

其参数含义如下。

- mode: 文件模式, 可选参数。
- bufsize: 缓冲区大小, 可选参数。

当完成通信后, 应使用 socket 对象的 close 方法关闭网络连接。

2. 建立服务端

使用 socket 模块建立一个简单的服务端。首先应创建一个 socket 对象, 使用 socket 对象的 bind 方法绑定 IP 地址和端口。然后使用 socket 对象的 listen 方法监听 socket 连接。最后进入循环等待客户端的连接。如下所示的 server.py 使用 socket 模块创建一个简单的服务端。

```
# -*- coding:utf-8 -*-
# file: server.py
#
import Tkinter
import threading
import socket

class ListenThread(threading.Thread):
    def __init__(self, edit, server):
        threading.Thread.__init__(self)
        self.edit = edit
        self.server = server
    def run(self):
        while 1:
            try:
                client, addr = self.server.accept()
                self.edit.insert(Tkinter.END,
                                '连接来自:%s:%d\n' % addr)
                data = client.recv(1024)
                self.edit.insert(Tkinter.END,
                                '收到数据:%s\n' % data)
                client.send('I GOT: %s' % data)
                client.close()
                self.edit.insert(Tkinter.END,
                                '关闭客户端\n')
            except:
                self.edit.insert(Tkinter.END,
                                '关闭连接\n')
                break

class Control(threading.Thread):
    def __init__(self, edit):
```

监听线程

保存窗口中的多行文本框

进入监听状态

使用 while 循环等待连接

捕获异常

等待连接

向文本框中输出状态

接收数据

向文本框中输出数据

发送数据

关闭同客户端的连接

向文本框中输出状态

异常处理

向文本框中输出状态

结束循环

控制线程

```

        threading.Thread._init_(self)
        self.edit = edit
        self.event = threading.Event()
        self.event.clear()
    def run(self):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('', 1051))
        server.listen(1)
        self.edit.insert(Tkinter.END, '正在等待连接\n')
        self.lt = ListenThread(self.edit, server)
        self.lt.setDaemon(True)
        self.lt.start()
        self.event.wait()
        server.close()
    def stop(self):
        self.event.set()
class Window:
    def _init_(self, root):
        self.root = root
        self.butlisten = Tkinter.Button(root,
            text = '开始监听', command = self.Listen)
        self.butlisten.place(x = 20, y = 15)
        self.butclose = Tkinter.Button(root,
            text = '停止监听', command = self.Close)
        self.butclose.place(x = 120, y = 15)
        self.edit = Tkinter.Text(root)
        self.edit.place(y = 50)
    def Listen(self):
        self.ctrl = Control(self.edit)
        self.ctrl.setDaemon(True)
        self.ctrl.start()
    def Close(self):
        self.ctrl.stop()
root = Tkinter.Tk()
window = Window(root)
root.mainloop()

```

保存窗口中的多行文本框
创建 Event 对象
清除 event 标志

创建 socket 连接
绑定本机 1051 端口
开始监听
向文本框中输出状态
创建监听线程对象

执行监听线程
进入等待状态
关闭连接
结束控制进程
设置 event 标志
主窗口

创建组件

处理按钮事件
创建控制线程对象

执行控制线程

结束控制线程

在 server.py 脚本中，由于使用 while 循环监听连接，为了避免图形界面下假死的状态，将 while 循环放在一个线程里执行，但 Python 中没有提供结束线程的函数或者方法，为了能随时终止监听，在脚本中创建了一个控制线程。通过控制线程执行监听线程，然后控制线程进入等待状态。当 event 被设置后，在控制线程中将关闭 socket 连接，监听也就停止了。由于监听线程已经进入监听状态，在控制线程中关闭 socket 连接将导致异常，所以在监听线程中使用 try 捕获异常，结束循环。

3. 建立客户端

客户端的创建相对简单，只要连接指定的 IP 和端口地址，然后向服务端发送数据即可，

如下所示的 client.py 脚本创建了一个简单的客户端。

```
# -*- coding:utf-8 -*-
# file: client.py
#
import Tkinter
import socket
class Window:
    def __init__(self, root):
        label1 = Tkinter.Label(root, text = 'IP')
        label2 = Tkinter.Label(root, text = 'Port')
        label3 = Tkinter.Label(root, text = 'Data')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        self.entryIP = Tkinter.Entry(root)
        self.entryIP.insert(Tkinter.END, '127.0.0.1')
        self.entryPort = Tkinter.Entry(root)
        self.entryPort.insert(Tkinter.END, '1051')
        self.entryData = Tkinter.Entry(root)
        self.entryData.insert(Tkinter.END, 'Hello')
        self.Recv = Tkinter.Text(root)
        self.entryIP.place(x = 40, y = 5)
        self.entryPort.place(x = 40, y = 30)
        self.entryData.place(x = 40, y = 55)
        self.Recv.place(y = 105)
        self.send = Tkinter.Button(root, text = '发送数据', command = self.Send)
        self.send.place(x = 40, y = 80)
    def Send(self):
        try:
            ip = self.entryIP.get()
            port = int(self.entryPort.get())
            data = self.entryData.get()
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client.connect((ip,port))
            client.send(data)
            rdata = client.recv(1024)
            self.Recv.insert(Tkinter.END, 'Server:' + rdata+ '\n')
            client.close()
        except :
            self.Recv.insert(Tkinter.END, '发送错误\n')
root = Tkinter.Tk()
window = Window(root)
root.mainloop()
```

创建组件

按钮事件

异常处理

获取 IP

获取端口

获取发送数据

创建 socket 对象

连接服务端

发送数据

接收数据

输出接收的数据

关闭连接

运行 server.py，单击【开始监听】按钮，进入监听状态，如图 18-1 所示。运行 client.py，单击【发送】按钮，可以向服务端发送数据，如图 18-2 所示。

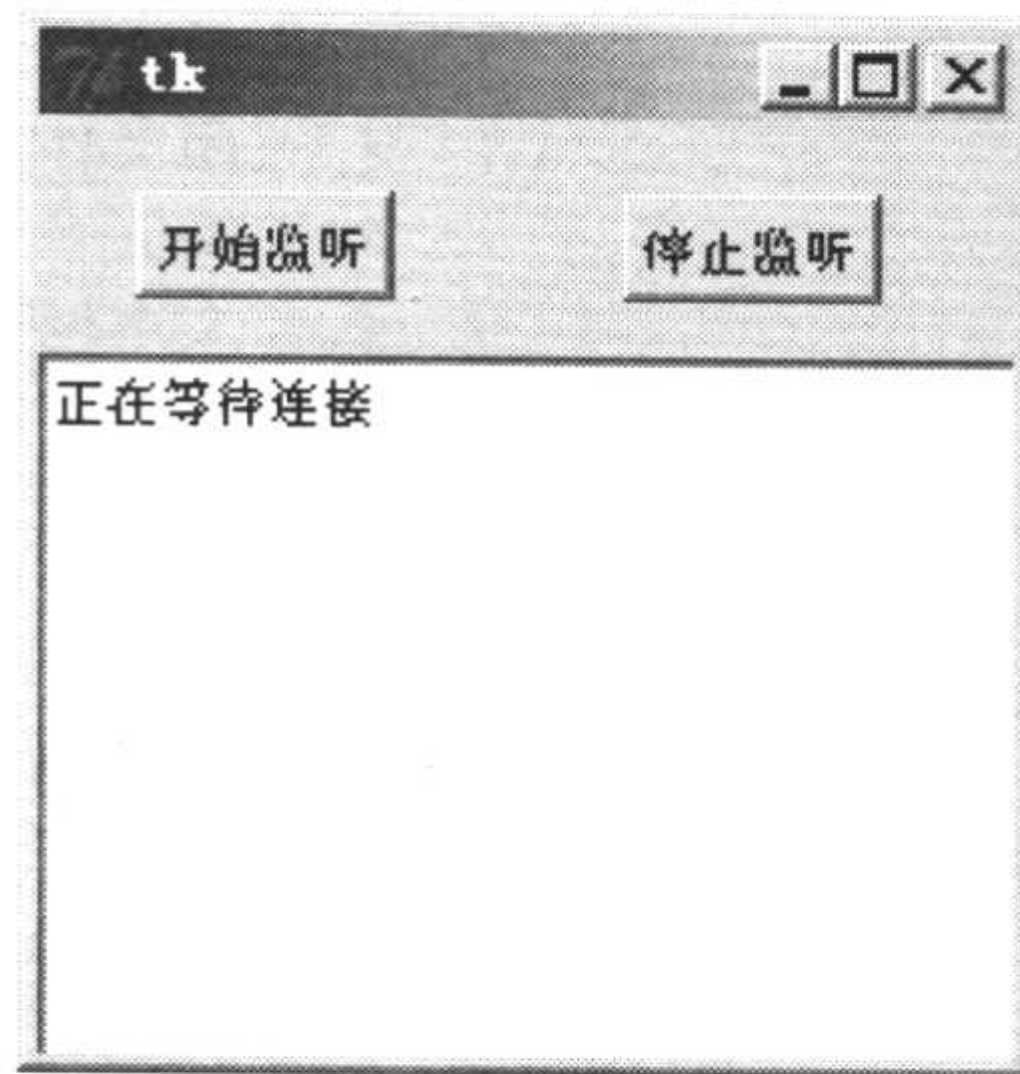


图 18-1 服务端



图 18-2 客户端

18.1.3 在局域网中传输文件

使用 Python 中的 socket 模块可以编写一个简单的传输文件的脚本。传输文件也就是将文件内容依次发送出去，因此可以修改上一节中的例子，将发送数据部分修改为发送文件的内容即可。如下所示的 FileServer.py 脚本修改了 server.py 脚本，用于接收文件。

```
# -*- coding:utf-8 -*-
# file: FileServer.py
#
import Tkinter
import threading
import socket
import os

class ListenThread(threading.Thread):                                # 创建监听线程
    def _init_(self, edit, server):
        threading.Thread._init_(self)
        self.edit = edit                                           # 保存窗口中的多行文本框
        self.server = server
        self.files = ['FileServer.py']

    def run(self):
        while 1:
            try:
                self.client, addr = self.server.accept()           # 进入监听状态
                self.edit.insert(Tkinter.END,                      # 使用 while 循环, 不停监听
                                '连接来自:%s:%d\n' % addr)         # 捕获异常
                data = self.client.recv(1024)                      # 等待连接
                self.edit.insert(Tkinter.END,                      # 向文本框中输出状态
                                '收到文件:%s \n' % data)           # 接收数据
                file = os.open(data, os.O_WRONLY|os.O_CREAT        # 向文本框中输出数据
                                |os.O_EXCL|os.O_BINARY)            # 创建文件
                while 1:
                    rdata = self.client.recv(1024)                 # 接收数据
                    if not rdata:
                        break
```



```

        os.write(file, rdata)
        os.close(file)
        self.client.close()
        self.edit.insert(Tkinter.END,
                          '关闭客户端\n')
    except:
        self.edit.insert(Tkinter.END,
                          '关闭连接\n')
        break
class Control(threading.Thread):
    def __init__(self, edit):
        threading.Thread.__init__(self)
        self.edit = edit
        self.event = threading.Event()
        self.event.clear()
    def run(self):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind(('', 1051))
        server.listen(1)
        self.edit.insert(Tkinter.END, '正在等待连接\n')
        self.lt = ListenThread(self.edit, server)
        self.lt.setDaemon(True)
        self.lt.start()
        self.event.wait()
        server.close()
    def stop(self):
        self.event.set()
class Window:
    def __init__(self, root):
        self.root = root
        self.butlisten = Tkinter.Button(root,
                                         text = '开始监听', command = self.Listen)
        self.butlisten.place(x = 20, y = 15)
        self.butclose = Tkinter.Button(root,
                                         text = '停止监听', command = self.Close)
        self.butclose.place(x = 120, y = 15)
        self.edit = Tkinter.Text(root)
        self.edit.place(y = 50)
    def Listen(self):
        self.ctrl = Control(self.edit)
        self.ctrl.setDaemon(True)
        self.ctrl.start()
    def Close(self):
        self.ctrl.stop()
root = Tkinter.Tk()
window = Window(root)
root.mainloop()

```

将数据写入文件
 # 关闭文件
 # 关闭同客户端的连接
 # 向文本框中输出状态
 # 异常处理
 # 向文本框中输出状态
 # 结束循环
 # 控制线程
 # 保存窗口中的多行文本框
 # 创建 Event 对象
 # 清除 event 标志
 # 创建 socket 连接
 # 绑定本机 1051 端口
 # 开始监听
 # 向文本框中输出状态
 # 创建监听线程对象
 # 执行监听线程
 # 进入等待状态
 # 关闭连接
 # 结束控制进程
 # 设置 event 标志
 # 主窗口
 # 创建组件
 # 处理按钮事件
 # 创建控制线程对象
 # 执行控制线程
 # 结束控制线程

如下所示的 FileClient.py 脚本修改了 client.py 脚本，用于发送文件（脚本中没有处理字

符编码，因此发送的文件路径和文件名不能包含中文字符)。

```
# -*- coding:utf-8 -*-
# file: FileClient.py
#
import Tkinter
import tkFileDialog
import socket
import os
class Window:
    def _init_(self, root):
        label1 = Tkinter.Label(root, text = 'IP')
        label2 = Tkinter.Label(root, text = 'Port')
        label3 = Tkinter.Label(root, text = '文件')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        self.entryIP = Tkinter.Entry(root)
        self.entryIP.insert(Tkinter.END, '127.0.0.1')
        self.entryPort = Tkinter.Entry(root)
        self.entryPort.insert(Tkinter.END, '1051')
        self.entryData = Tkinter.Entry(root)
        self.entryData.insert(Tkinter.END, 'Hello')
        self.entryIP.place(x = 40, y = 5)
        self.entryPort.place(x = 40, y = 30)
        self.entryData.place(x = 40, y = 55)
        self.send = Tkinter.Button(root, text = '发送文件', command = self.Send)
        self.openfile = Tkinter.Button(root, text = '浏览', command = self.Openfile)
        self.send.place(x = 40, y = 80)
        self.openfile.place(x = 170, y = 55)
    def Send(self):
        try:
            ip = self.entryIP.get()
            port = int(self.entryPort.get())
            filename = self.entryData.get()
            tt = filename.split('/')
            name = tt[len(tt)-1]
            client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            client.connect((ip,port))
            client.send(name)
            file = os.open(filename,
                           os.O_RDONLY | os.O_EXCL|os.O_BINARY)
            while 1:
                data = os.read(file,1024)
                if not data:
                    break
                client.send(data)
            os.close(file)
            client.close()
        except :
```

创建组件

按钮事件

异常处理

获取 IP

获取端口

获取发送数据

创建 socket 对象

连接服务端

发送数据

打开文件

发送文件

关闭文件

关闭连接


```

        print '发送错误'
    def Openfile(self):
        r = tkFileDialog.askopenfilename(title = 'Python Tkinter', # 创建打开文件对话框
            filetypes=[('All files', '*'), ('Python', '*.py *.pyw')])
        if r:
            self.entryData.delete(0, Tkinter.END)
            self.entryData.insert(Tkinter.END, r)
root = Tkinter.Tk()
window = Window(root)
root.mainloop()

```

运行 FileServer.py 脚本如图 18-3 所示, 运行 FileClient.py 脚本如图 18-4 所示。

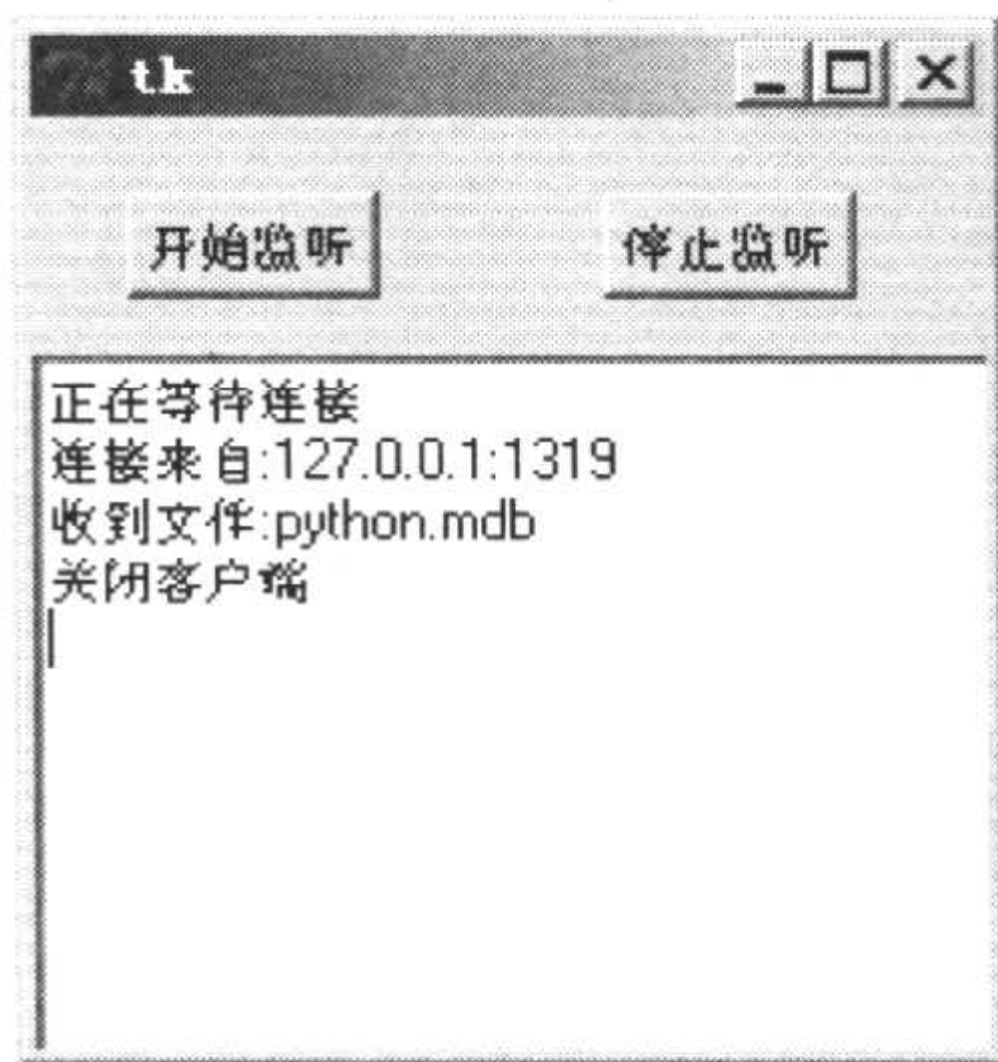


图 18-3 FileServer 接收文件

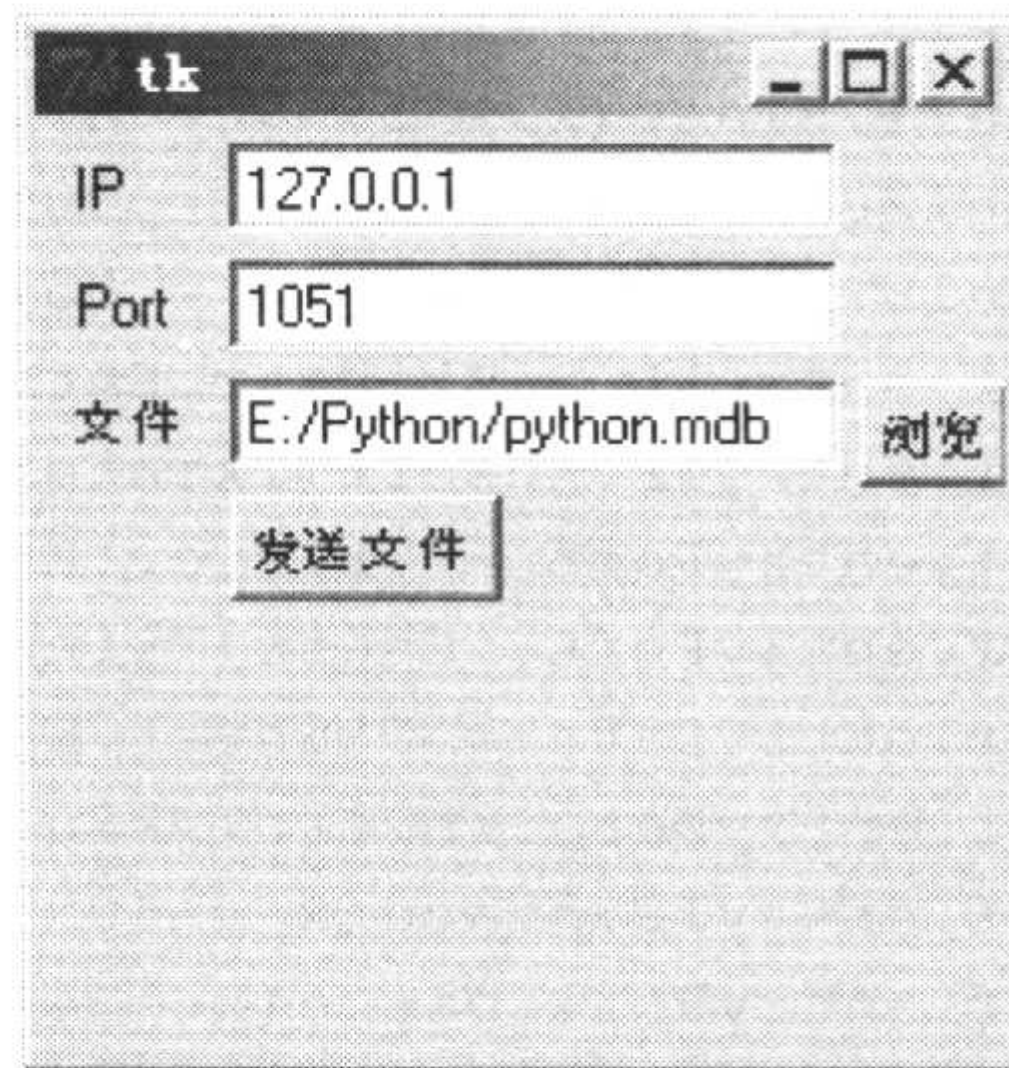


图 18-4 FileClient 发送文件

18.2 使用 urllib、httplib 以及 ftplib

Python 提供的 socket 的模块主要用于底层网络协议, 对于常用的 HTTP 协议和 FTP 协议可以使用 Python 中的 httplib 和 ftplib 进行访问。

18.2.1 使用 Python 访问网站

网站都是基于 HTTP 协议的, 使用 Python 中的 urllib 和 httplib 都可以访问网站。其中 urllib 主要用于处理 URL (Universal Resource Locator), 使用 urllib 操作 URL 可以使用和打开本地文件一样的操作。而 httplib 则实现了对 HTTP 协议的封装。

1. urllib 模块简介

使用 Python 中的 urllib 模块可以对 URL 进行处理。使用 urllib 模块中的 urlopen 函数可以打开一个 URL。其原型如下所示。

```
urlopen(url, data, proxies)
```

其参数含义如下。

- url: 要进行操作的 URL 地址。

- data: 向 URL 传递的数据, 可选参数。
- proxies: 使用的代理地址, 可选参数。

urlopen 将返回一个类似于 file 的对象, 可以像操作文件一样使用 read、readline、close 等方法对 URL 进行操作。使用 urllib 模块中的 urlretrieve 可以将 URL 保存为本地文件。

```
urlretrieve(url, filename, reporthook, data)
```

其参数含义如下。

- url: 要保存的 URL 地址。
- filename: 指定保存的文件名, 可选参数。
- reporthook: 回调函数, 可选参数。
- data: 发送的数据, 一般用于 POST, 可选参数。

使用 urllib 模块中的 urlencode 可以对 URL 进行编码, 其原型如下所示。

```
urlencode(query, doseq)
```

其参数含义如下。

- query: 由要进行编码的变量和值组成的字典。
- doseq: 可选参数, 若为 True, 则将为元组的值分别编码成“变量=值”的形式。

使用 urllib 模块中的 quote 和 quote_plus 可以替换字符串中的特殊字符, 使其符合 URL 所要求使用的字符, 其原型分别如下所示。

```
quote(string, safe)
quote_plus(string, safe)
```

其参数含义如下。

- string: 要进行替换的字符串。
- safe: 可选参数, 指定不需要替换的字符, 默认为“/”。

使用 urllib 模块中的 unquote 和 unquote_plus 可以将使用 quote 和 quote_plus 替换后的字符还原, 其原型分别如下所示。

```
unquote(string)
unquote_plus(string)
```

其参数含义如下。

- string: 要进行还原的字符串。

2. httplib 模块简介

在 Python 的 httplib 模块中提供了 HTTPConnection 对象和 HTTPResponse 对象。当创建一个 HTTPConnection 对象后, 可以使用 request 方法向服务器发送请求, 其原型如下所示。

```
request(method, url, body, headers)
```

其参数含义如下。

- method: 发送的操作, 一般为“GET”或“POST”。

- url: 进行操作的 URL。
- body: 发送的数据。
- headers: 发送的 HTTP 头。

当向服务器发送请求后, 可以使用 `HTTPConnection` 对象的 `getresponse` 方法返回一个 `HTTPResponse` 对象。使用 `HTTPConnection` 对象的 `close` 方法可以关闭同服务器的连接。除了使用 `request` 方法以外, 还可以依次使用如下所示的方法向服务器发送请求。

```
putrequest(request, selector, skip_host, skip_accept_encoding)
putheader(header, argument, ...)
endheaders()
send(data)
```

对于 `putrequest`, 其参数含义如下。

- request: 所发送的操作。
- selector: 进行操作的 URL。
- skip_host: 可选参数, 若为真, 禁止自动发送 “HOST:”。
- skip_accept_encoding: 可选参数, 若为真, 禁止自动发送 “Accept-Encoding: headers”。

对于 `putheader`, 其参数含义如下。

- header: 发送的 HTTP 头。
- argument: 发送的参数。

对于 `send`, 其参数含义如下。

- data: 发送的数据。

`urllib` 模块中的 `HTTPResponse` 对象主要用于处理服务器对所发送请求的响应。使用 `HTTPResponse` 对象的 `read` 方法可以获得服务器响应主体。使用 `HTTPResponse` 对象的 `getheader` 可以获得服务器响应的 HTTP 头, 其原型如下所示。

```
getheader(name, default)
```

其参数含义如下。

- name: 指定 HTTP 头名。
- default: 可选参数, 如果指定是 name 不存在, 则获取 default 指定的 HTTP 头。

`HTTPResponse` 对象还具有 `version`、`status` 和 `reason` 等属性, 用于查看 HTTP 协议的版本、状态等。

3. 使用 Python 访问网站

使用 Python 的 `urllib` 模块可以创建一个简单的访问网站的脚本, 获取指定的页面。如果使用 GUI 库中显示 HTML 的组件, 还可以制作一个简单的 Python Web 浏览器。使用 `httplib` 模块也可以访问网站, 但过程比 `urllib` 模块要复杂。`httplib` 模块可以用于需要用户名和密码

认证的网站，而 urllib 模块则只能简单地访问、下载页面内容。如下所示的 httpurl.py 脚本使用 urllib 模块读取网页内容。

```
# -*- coding:utf-8 -*-
# file: httpurl.py
#
import Tkinter
import urllib
class Window:
    def _init_(self, root):
        self.root = root
        self.entryUrl = Tkinter.Entry(root)           # 创建组件
        self.entryUrl.place(x = 5, y = 15)
        self.get = Tkinter.Button(root,
            text = '下载页面', command = self.Get)
        self.get.place(x = 120, y = 15)
        self.edit = Tkinter.Text(root)
        self.edit.place(y = 50)
    def Get(self):
        url = self.entryUrl.get()                       # 获取 URL
        page = urllib.urlopen(url)                     # 打开 URL
        data = page.read()                             # 读取 URL 内容
        self.edit.insert(Tkinter.END, data)            # 将内容输出到文本框
        page.close()
root = Tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.mainloop()
```

运行 httpurl.py 后，输入 <http://www.python.org>，单击下载页面，如图 18-5 所示。

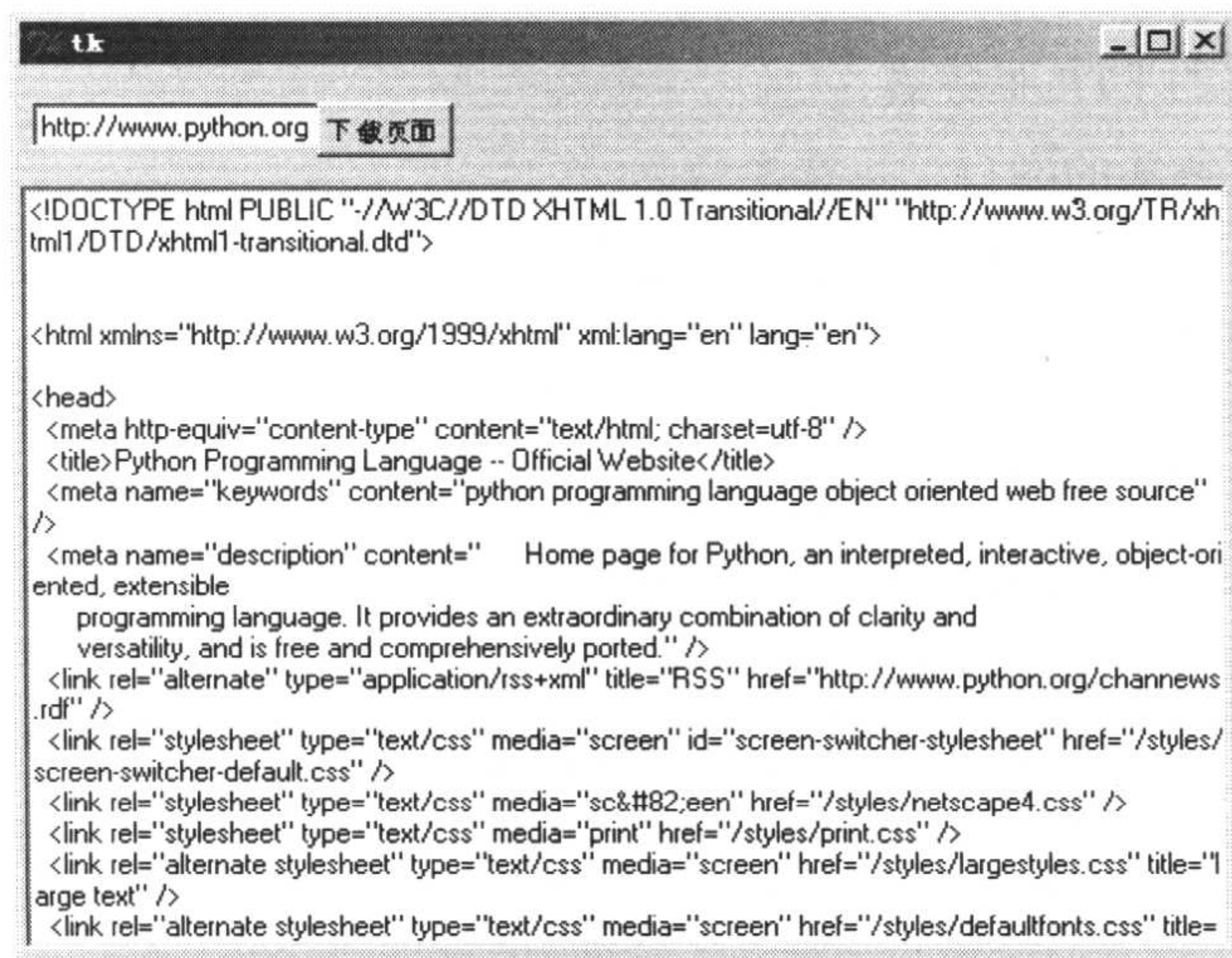


图 18-5 使用 urllib 下载页面

18.2.2 访问 FTP

Python 中的 `ftplib` 模块提供了用于访问 FTP 的函数。使用 `ftplib` 模块可以在 Python 脚本中访问 FTP，完成上传、下载文件等。

1. `ftplib` 模块简介

使用 `ftplib` 模块中的 `FTP` 类，可以创建一个 FTP 连接对象。其原型如下所示。

```
FTP(host, user, passwd, acct)
```

其参数含义如下。

- `host`: 要连接的 FTP 服务器，可选参数。
- `user`: 登录 FTP 服务器所使用的用户名，可选参数。
- `passwd`: 登录 FTP 服务器所使用的密码，可选参数。
- `acct`: 可选参数，默认为空。

当创建一个 FTP 连接对象以后，可以使用 `set_debuglevel` 方法设置调试级别。其原型如下所示。

```
set_debuglevel(level)
```

其参数含义如下。

- `level`: 调试级别，默认的调试级别为 0。

如果在创建 FTP 连接对象时未使用 `HOST` 参数，则可以使用 FTP 对象的 `connect` 方法，其原型如下所示。

```
connect(host, port)
```

其参数含义如下。

- `host`: 要连接的 FTP 服务器。
- `port`: FTP 服务器的端口，可选参数。

如果在创建 FTP 对象时未使用用户名和密码，则可以使用 FTP 对象的 `login` 对象使用用户名和密码登录到 FTP 服务器。其原型如下所示。

```
login(user, passwd, acct)
```

- `user`: 登录 FTP 服务器所使用的用户名。
- `passwd`: 登录 FTP 服务器所使用的密码。
- `acct`: 可选参数，默认为空。

使用 FTP 对象的 `getwelcome` 方法可以获得 FTP 服务器的欢迎信息。使用 FTP 对象的 `abort` 方法可以中断文件传输。使用 FTP 对象的 `sendcmd` 和 `voidcmd` 方法可以向 FTP 服务器发送命令。不同的是 `voidcmd` 没有返回值。其函数原型分别如下所示。

```
sendcmd(command)
voidcmd(command)
```

其参数含义如下。

- **command**: 向服务器发送的命令字符串。

使用 FTP 对象的 `retrbinary` 和 `retrlines` 方法可以从 FTP 服务器下载文件。不同的是 `retrbinary` 方法使用二进制形式传输文件，而 `retrlines` 方法使用 ASCII 形式传输文件。其函数原型分别如下所示。

```
retrbinary(command, callback, maxblocksize, rest)
retrlines(command, callback)
```

对于 `retrbinary`，其参数含义如下。

- **command**: 传输命令，由“RETR+文件名”组成（之间有空格）。
- **callback**: 传输回调函数。
- **maxblocksize**: 设置每次传输的最大字节数，可选参数。
- **rest**: 设置文件续传位置，可选参数。

对于 `retrlines`，其参数含义如下所示。

- **command**: 传输命令。
- **callback**: 传输回调函数。

使用 FTP 对象的 `storbinary` 和 `storlines` 方法可以向 FTP 服务器上传文件。不同的是 `storbinary` 方法使用二进制形式传输文件，而 `storlines` 方法使用 ASCII 形式传输文件。其函数原型分别如下所示。

```
storbinary(command, file, blocksize)
storlines(command, file)
```

对于 `storbinary`，其参数含义如下。

- **command**: 传输命令，由“STOR+文件名”组成（之间有空格）。
- **file**: 本地文件句柄。
- **blocksize**: 设置每次读取文件最大字节数，可选参数。

对于 `storlines`，其参数含义如下。

- **command**: 传输命令。
- **file**: 本地文件句柄。

使用 FTP 对象的 `set_pasv` 方法可以设置传输模式。其函数原型如下所示。

```
set_pasv(boolean)
```

其参数含义如下。

- **boolean**: 如果为 `True`，则为被动模式；如果为 `False`，则为主动模式。

使用 FTP 对象的 `dir` 方法可以获取当前目录中的内容列表。使用 FTP 对象的 `rename` 方法可以修改 FTP 服务器中的文件名。其原型如下所示。


```
rename(fromname, toname)
```

其参数含义如下。

- fromname: 原来文件名。
- toname: 重命名后的文件名。

使用 FTP 对象的 delete 方法可以从 FTP 服务器上删除文件。其原型如下所示。

```
delete(filename)
```

其参数含义如下。

- filename: 要删除的文件名。

使用 FTP 对象的 cwd 方法可以改变当前目录。其原型如下所示。

```
cwd(pathname)
```

其参数含义如下。

- pathname: 要进入目录的路径。

使用 FTP 对象的 mkd 方法可以在 FTP 服务器上创建目录。其原型如下所示。

```
mkd(pathname)
```

其参数含义如下。

- pathname: 要创建目录的路径。

使用 FTP 对象的 pwd 方法可以获得当前目录。使用 FTP 对象的 rmd 方法可以删除 FTP 服务器上的目录。其原型如下所示。

```
rmd(dirname)
```

其参数含义如下。

- dirname: 要删除的目录。

使用 FTP 对象的 size 方法可以获得文件的大小。其原型如下所示。

```
size(filename)
```

其参数含义如下。

- filename: 文件名。

使用 FTP 对象的 quit 和 close 方法可以关闭同 FTP 服务器的连接。

2. 使用 Python 访问 FTP

Python 的 ftplib 模块提供了完整的用于 FTP 协议的函数、方法，使用 ftplib 模块可以制作一个简单的类似于 Windows 自带的 FTP 客户端。如下所示的 pyftp.py 使用 ftplib 模块创建了一个简单的 FTP 客户端。

```
# -*- coding:utf-8 -*-
# file: pyftp.py
#
import string
from ftplib import FTP
```

```
# 从 ftplib 模块中导入 FTP
```

```

bufsize = 1024                                     # 设置缓冲区大小
def Get(filename):                                  # 下载文件
    command = 'RETR ' + filename
    ftp.retrbinary(command, open(filename, 'wb').write, bufsize)
    print '下载成功'
def Put(filename):                                  # 上传文件
    command = 'STOR ' + filename
    filehandler = open(filename, 'rb')
    ftp.storbinary(command, filehandler, bufsize)
    filehandler.close()
    print '上传成功'
def PWD():                                          # 获取当前目录
    print ftp.pwd()
def Size(filename):                                # 获取文件大小
    print ftp.size(filename)
def Help():                                         # 输出帮助
    print '''
=====
    Simple Python FTP
=====
cd          进入文件夹
delete      删除文件
dir         获取当前文件列表
get         下载文件
help        帮助
mkdir       创建文件夹
put         上传文件
pwd         获取当前目录
rename      重命名文件
rmdir       删除文件夹
size        获取文件大小
'''

server = raw_input('请输入 FTP 服务器地址:')      # 获取服务器地址
ftp = FTP(server)                                  # 连接到服务器地址
username = raw_input('请输入用户名:')              # 获取用户名
password = raw_input('请输入密码:')                # 获取字典
ftp.login(username, password)                      # 登录 FTP
print ftp.getwelcome()                             # 获取欢迎信息
actions = {'dir':ftp.dir, 'pwd': PWD, 'cd':ftp.cwd, 'get':Get, # 命令与对应的函数字典
          'put':Put, 'help':Help, 'rmdir': ftp.rmd,
          'mkdir': ftp.mkd, 'delete':ftp.delete,
          'size':Size, 'rename':ftp.rename}

while True:                                        # 命令循环
    print 'pyftp>',                                # 输出提示符
    cmds = raw_input()                             # 获取输入
    cmd = string.split(cmds)                        # 将输入按空格分割
    try:                                             # 异常处理
        if len(cmd) == 1:                           # 判断命令是否有参数
            if string.lower(cmd[0]) == 'quit':        # 如果命令为 quit,

```



```

        break
    else:
        actions[string.lower(cmd[0])]()
    elif len(cmd) == 2:
        actions[string.lower(cmd[0])](cmd[1])
    elif len(cmd) == 3:
        actions[string.lower(cmd[0])](cmd[1],cmd[2])
    else:
        print '输入错误'
except:
    print '命令出错'
ftp.quit()

```

则退出循环

调用与命令对应的函数
处理命令有一个参数的情况
调用与命令对应的函数
处理命令有两个参数的情况
调用与命令对应的函数

端口连接

18.3 使用 poplib 和 smtplib 模块收发邮件

Python 中的 poplib 模块和 smtplib 模块提供了对 POP3 协议和 SMTP 协议的支持。使用 POP3 协议可以登录 E-mail 收取邮件，使用 SMTP 协议可以发送邮件。

18.3.1 检查 E-mail

一般的邮箱服务都提供了 POP3 收取邮件的方式。Outlook 等 E-mail 客户端就是使用 POP3 协议收取邮箱中的邮件。使用 Python 的 poplib 模块可以实现一个简单的收取邮件的客户端。

1. poplib 模块简介

使用 poplib 模块中的 POP3 类可以创建一个 POP3 对象实例。其原型如下所示。

POP3(host, port)

其参数含义如下。

- host: POP3 邮件服务器。
- port: 服务器端口，可选参数，默认为 110。

当创建一个 POP3 对象实例后可以使用其 user 方法向 POP3 服务器发送用户名。其原型如下所示。

user(username)

其参数含义如下。

- username: 登录服务器的用户名。

使用 POP3 对象的 pass_方法可以向 POP3 服务器发送密码。其原型如下所示。

pass_(password)

其参数含义如下。

- password: 登录服务器的密码。

当登录服务器后可以使用 POP3 对象的 getwelcome 方法获取服务器的欢迎信息。使用

POP3 对象的 `set_debuglevel` 方法可以设置调试级别。其原型如下所示。

```
set_debuglevel(level)
```

其参数含义如下。

- `level`: 调试级别。

使用 POP3 对象的 `stat` 可以获取邮箱的状态, 如邮件数、邮箱大小等。使用 POP3 对象的 `list` 方法可以获得邮件的内容列表, 其原型如下所示。

```
list(which)
```

其参数含义如下。

- `which`: 可选参数, 如果指定, 则仅列出指定的邮件内容。

使用 POP3 对象的 `retr` 方法可以获取指定的邮件。其原型如下所示。

```
retr(which)
```

其参数含义如下。

- `which`: 指定要获取的邮件。

使用 POP3 对象的 `dele` 方法可以删除指定的邮件。其原型如下所示。

```
dele(which)
```

其参数含义如下。

- `which`: 指定要删除的邮件。

使用 POP3 对象的 `top` 方法可以收取邮件的部分内容。其原型如下所示。

```
top(which, howmuch)
```

其参数含义如下。

- `which`: 指定获取的邮件。
- `howmuch`: 指定获取的行数。

使用 POP3 对象的 `rset` 方法可以清除收件箱中邮件的删除标记。使用 POP3 对象的 `noop` 方法可以保持同服务器的连接。使用 POP3 对象的 `quit` 方法可以断开同服务器的连接。

2. 使用 Python 收取 E-mail

使用 Python 检查 E-mail 首先应该知道自己所使用的 E-mail 的 POP3 服务器地址和端口。对于网易 163 的邮箱, 其 POP3 服务器的地址为 `pop.163.com`, 端口为默认值 110。对于网易 126 的邮箱, 其 POP3 服务器地址为 `pop.126.com`, 端口为默认值 110。使用其他 E-mail 可以查看网站帮助, 获取 POP3 服务器的地址和端口。如下所示的 `pypop.py` 脚本可以将邮箱中的邮件下载到本地 (使用时应注意, 可以申请一个测试邮箱测试该脚本, 避免误操作导致邮箱中邮件损失)。

```
# -*- coding:utf-8 -*-
# file: pypop.py
#
import poplib
```



```

import re
import Tkinter
class Window:
    def _init_(self, root):
        label1 = Tkinter.Label(root, text = 'POP3')
        label2 = Tkinter.Label(root, text = 'Port')
        label3 = Tkinter.Label(root, text = '用户名')
        label4 = Tkinter.Label(root, text = '密码')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        label4.place(x = 5, y = 80)
        self.entryPOP = Tkinter.Entry(root)
        self.entryPort = Tkinter.Entry(root)
        self.entryUser = Tkinter.Entry(root)
        self.entryPass = Tkinter.Entry(root, show = '*')
        self.entryPort.insert(Tkinter.END, '110')
        self.entryPOP.place(x = 40, y = 5)
        self.entryPort.place(x = 40, y = 30)
        self.entryUser.place(x = 40, y = 55)
        self.entryPass.place(x = 40, y = 80)
        self.get = Tkinter.Button(root, text = '收取邮件', command = self.Get)
        self.get.place(x = 60, y = 120)
        self.text = Tkinter.Text(root)
        self.text.place(y=150)
    def Get(self):
        try:
            host = self.entryPOP.get()
            port = int(self.entryPort.get())
            user = self.entryUser.get()
            pw = self.entryPass.get()
            pop = poplib.POP3(host)
            pop.user(user)
            pop.pass_(pw)
            stat = pop.stat()
            self.text.insert(Tkinter.END,
                'Status: %d message(s), %d bytes\n' % stat)
            rx_headers = re.compile(r"^(From|To|Subject)")
            for n in range(stat[0]):
                response, lines, bytes = pop.top(n + 1, 10)
                self.text.insert(Tkinter.END,
                    "Message %d (%d bytes)\n" % (n+1, bytes))
                self.text.insert(Tkinter.END, "-" * 30 + '\n')
                self.text.insert(Tkinter.END,
                    "\n".join(filter(rx_headers.match, lines)))
                self.text.insert(Tkinter.END, '\n')
                self.text.insert(Tkinter.END, "-" * 30 + '\n')
            except :
                self.text.insert(Tkinter.END, '接受错误\n')

```

创建组件

按钮事件

异常处理

获取服务器地址

获取端口

获取用户名

获取密码

创建 POP3 实例

登录服务器

获取状态

输出状态

编译正则表达式

收取邮件的前 10 行

输出匹配到的内容

```
root = Tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.mainloop()
```

18.3.2 发送 E-mail

发送邮件一般使用的是 SMTP 协议，使用 Python 的 smtplib 模块可以登录 SMTP 协议发送邮件。使用 SMTP 协议发送邮件，首先要登录 SMTP 服务器。

1. smtplib 模块简介

使用 smtplib 模块的 SMTP 类可以创建一个 SMTP 对象实例。其原型如下所示。

```
SMTP(host, port, local_hostname)
```

其参数含义如下。

- host: 连接的服务器名，可选参数。
- port: 服务器端口，可选参数。
- local_hostname: 本地主机名，可选参数。

如果在创建 SMTP 对象时没有指定 host 和 port，可以使用 SMTP 对象的 connect 方法连接到服务器。其原型如下所示。

```
connect(host, port)
```

其参数含义如下。

- host: 连接的服务器名，可选参数。
- port: 服务器端口，可选参数。

使用 SMTP 对象的 login 方法可以使用用户名和密码登录到 SMTP 服务器。其原型如下所示。

```
login(user, password)
```

其参数含义如下。

- user: 登录服务器的用户名。
- password: 登录服务器的密码。

使用 SMTP 对象的 set_debuglevel 方法可以设置调试级别。其原型如下所示。

```
set_debuglevel(level)
```

其参数含义如下。

- level: 调试级别。

使用 SMTP 对象的 docmd 方法可以向 SMTP 服务器发送命令。其原型如下所示。

```
docmd(cmd, , argstring)
```

其参数含义如下。

- cmd: 向 SMTP 服务器发送的命令。

- argstring: 命令参数, 可选参数。

使用 SMTP 对象的 sendmail 方法可以发送邮件。其原型如下所示。

```
sendmail(from_addr, to_addrs, msg, mail_options, rcpt_options)
```

其参数含义如下。

- from_addr: 发送者邮件地址。
- to_addrs: 邮件发送地址。
- msg: 邮件内容。
- mail_options: 可选参数, 邮件 ESMTP 操作。
- rcpt_options: 可选参数, RCPT 操作。

使用 SMTP 对象的 quit 方法可以断开同服务器的连接。

2. 使用 Python 发送邮件

和使用 Python 接收 E-mail 一样, 使用 Python 发送 E-mail 时, 也应该找到所使用 E-mail 的 SMTP 服务器的地址和端口。对于网易 163 的邮箱, 其 SMTP 服务器的地址为 smtp.163.com, 端口为默认值 25。对于网易 126 的邮箱, 其 SMTP 服务器的地址为 smtp.126.com, 端口为默认值 25。如下所示的 pysmtp.py 脚本使用 Python 的 smtplib 模块发送邮件 (为了简便起见, 没有对字符进行编码, 所以不能发送中文字符)。

```
# -*- coding:utf-8 -*-
# file: pysmtp.py
#
import smtplib
import Tkinter

class Window:
    def __init__(self, root):
        # 创建组件
        label1 = Tkinter.Label(root, text = 'SMTP')
        label2 = Tkinter.Label(root, text = 'Port')
        label3 = Tkinter.Label(root, text = '用户名')
        label4 = Tkinter.Label(root, text = '密码')
        label5 = Tkinter.Label(root, text = '收件人')
        label6 = Tkinter.Label(root, text = '主题')
        label7 = Tkinter.Label(root, text = '发件人')
        label1.place(x = 5, y = 5)
        label2.place(x = 5, y = 30)
        label3.place(x = 5, y = 55)
        label4.place(x = 5, y = 80)
        label5.place(x = 5, y = 105)
        label6.place(x = 5, y = 130)
        label7.place(x = 5, y = 155)
        self.entryPOP = Tkinter.Entry(root)
        self.entryPort = Tkinter.Entry(root)
        self.entryUser = Tkinter.Entry(root)
        self.entryPass = Tkinter.Entry(root, show = '*')
```

```

self.entryTo = Tkinter.Entry(root)
self.entrySub = Tkinter.Entry(root)
self.entryFrom = Tkinter.Entry(root)
self.entryPort.insert(Tkinter.END, '25')
self.entryPOP.place(x = 50, y = 5)
self.entryPort.place(x = 50, y = 30)
self.entryUser.place(x = 50, y = 55)
self.entryPass.place(x = 50, y = 80)
self.entryTo.place(x = 50, y = 105)
self.entrySub.place(x = 50, y = 130)
self.entryFrom.place(x = 50, y = 155)
self.get = Tkinter.Button(root, text = '发送邮件', command = self.Get)
self.get.place(x = 60, y = 180)
self.text = Tkinter.Text(root)
self.text.place(y=200)
def Get(self):
    try:
        host = self.entryPOP.get()
        port = int(self.entryPort.get())
        user = self.entryUser.get()
        pw = self.entryPass.get()
        fromaddr = self.entryFrom.get()
        toaddr = self.entryTo.get()
        subject = self.entrySub.get()
        text = self.text.get(1.0, Tkinter.END)
        msg = ("From: %s\nTo: %s\nSubject: %s\n\n"
              % (fromaddr, toaddr, subject))
        msg = msg + text
        smtp = smtplib.SMTP(host,port)
        smtp.set_debuglevel(1)
        smtp.login(user,pw)
        smtp.sendmail(fromaddr, toaddr, msg)
        smtp.quit()
    except :
        self.text.insert(Tkinter.END, '发送错误\n')
root = Tkinter.Tk()
window = Window(root)
root.mainloop()

```

```

# 按钮事件
# 异常处理
# 获取服务器地址
# 获取端口
# 获取用户名
# 获取密码
# 获取发件人
# 获取收件人
# 获取主题
# 获取邮件内容
# 生成邮件头

# 连接服务器
# 设置调试级别
# 登录服务器
# 发送邮件
# 断开服务器

```

18.4 连接到 Gtalk

现在流行的 Gtalk 是基于 XMPP 开放协议的即时通信工具。由于使用的是开放协议，只要遵循 XMPP 协议，就可以连接到 Gtalk 服务器。也就是可以使用任意支持 XMPP 协议的客户端连接到 Gtalk。而类似于 QQ 等的即时通信工具使用的是自己的协议，因此只能使用腾讯提供的 QQ 客户端，而不能使用其他的客户端。XMPPY 模块是使用 Python 编写的支持 XMPP 协议的模块，使用 XMPPY 模块，可以使用 Python 编写一个简单的 Gtalk 客户端，或者一

个简单的聊天机器人。

18.4.1 安装 XMPPPY

由于 XMPPPY 模块依赖于 dnspython，因此应首先从 dnspython 官方网站 <http://www.dnspython.org> 下载 dnspython 在 Windows 下的安装程序。以 Python2.5 为例，从 dnspython 官方网站下载 dnspython-1.5.0.win32.exe 安装程序，其安装过程如下所示。

(1) 双击运行安装程序，如图 18-6 所示。

(2) 单击【下一步】按钮，进入安装选择界面，如图 18-7 所示。根据所安装 Python 的版本，选择安装路径。如果安装多个版本的 Python，则需要选择使用的版本，此处选择 Python 2.5。

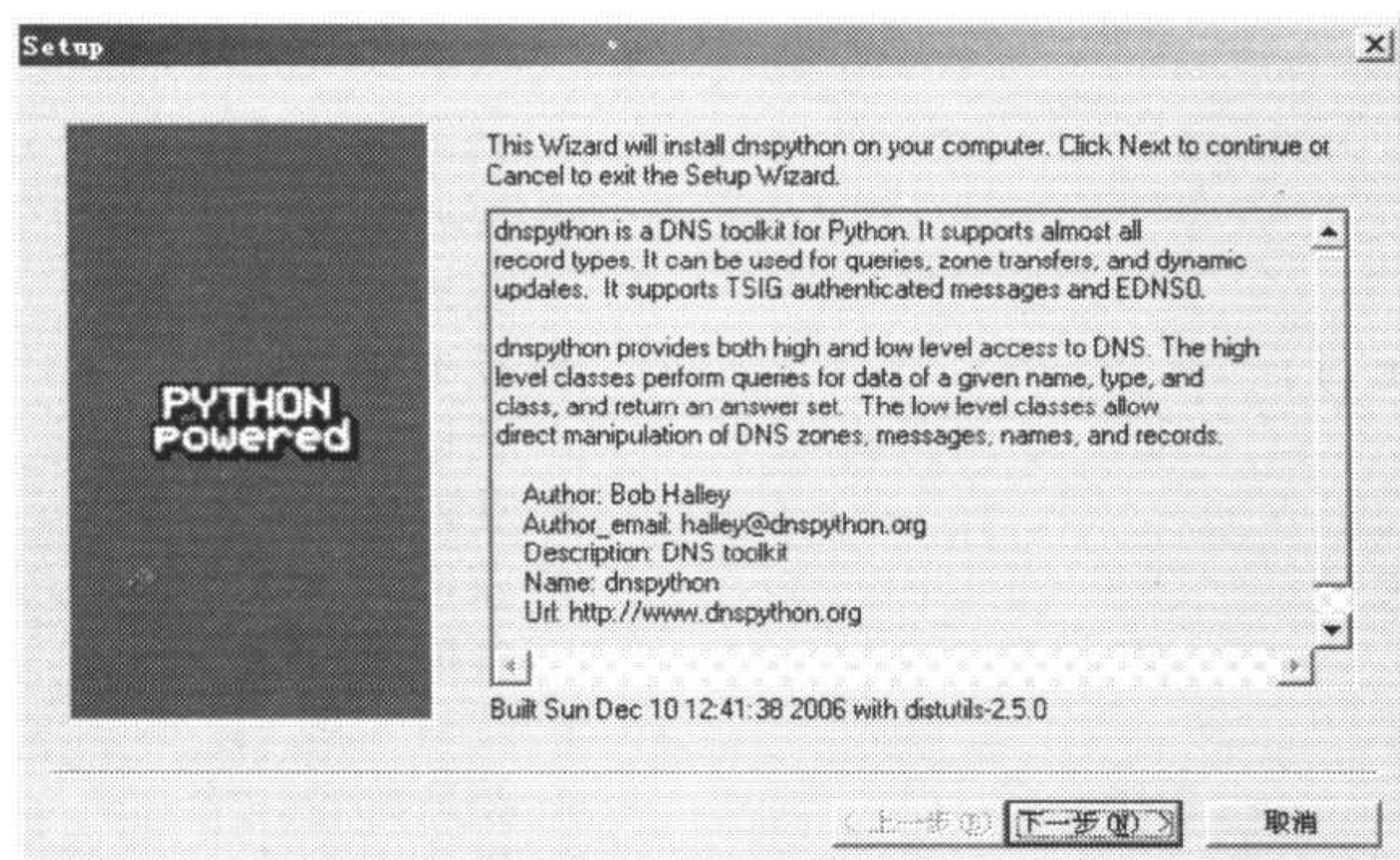


图 18-6 dnspython 安装界面

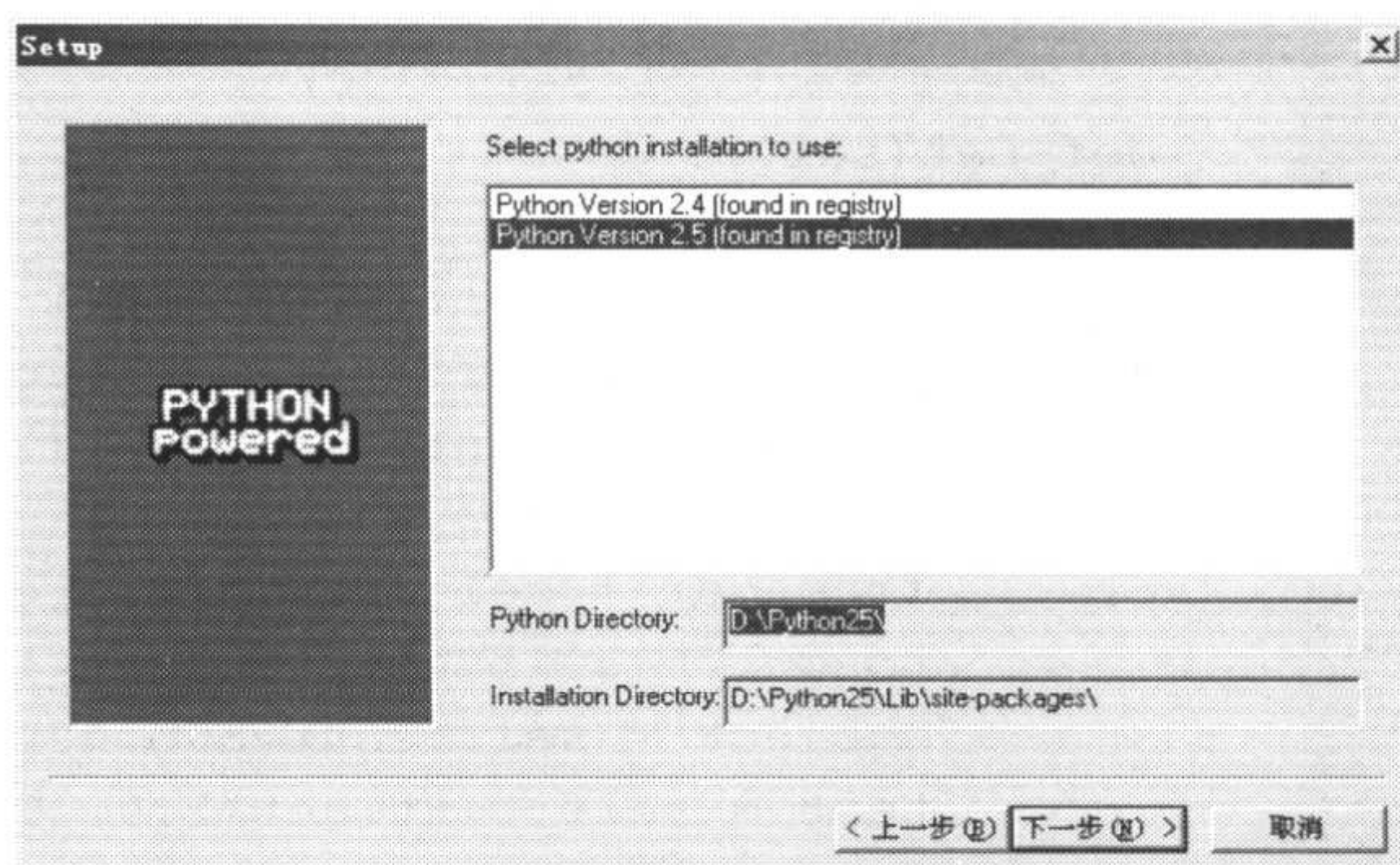


图 18-7 选择安装路径

(3) 单击【下一步】按钮，进入确认安装界面，如图 18-8 所示。单击【下一步】按钮，可以完成 dnspython 的安装。

XMPPPY 可以从其官方网站 <http://xmpppy.sourceforge.net> 下载 xmpppy-0.4.0.win32.exe 安装程序，以 Python2.5 为例，其安装过程如下所示。

(1) 双击运行安装程序，如图 18-9 所示。

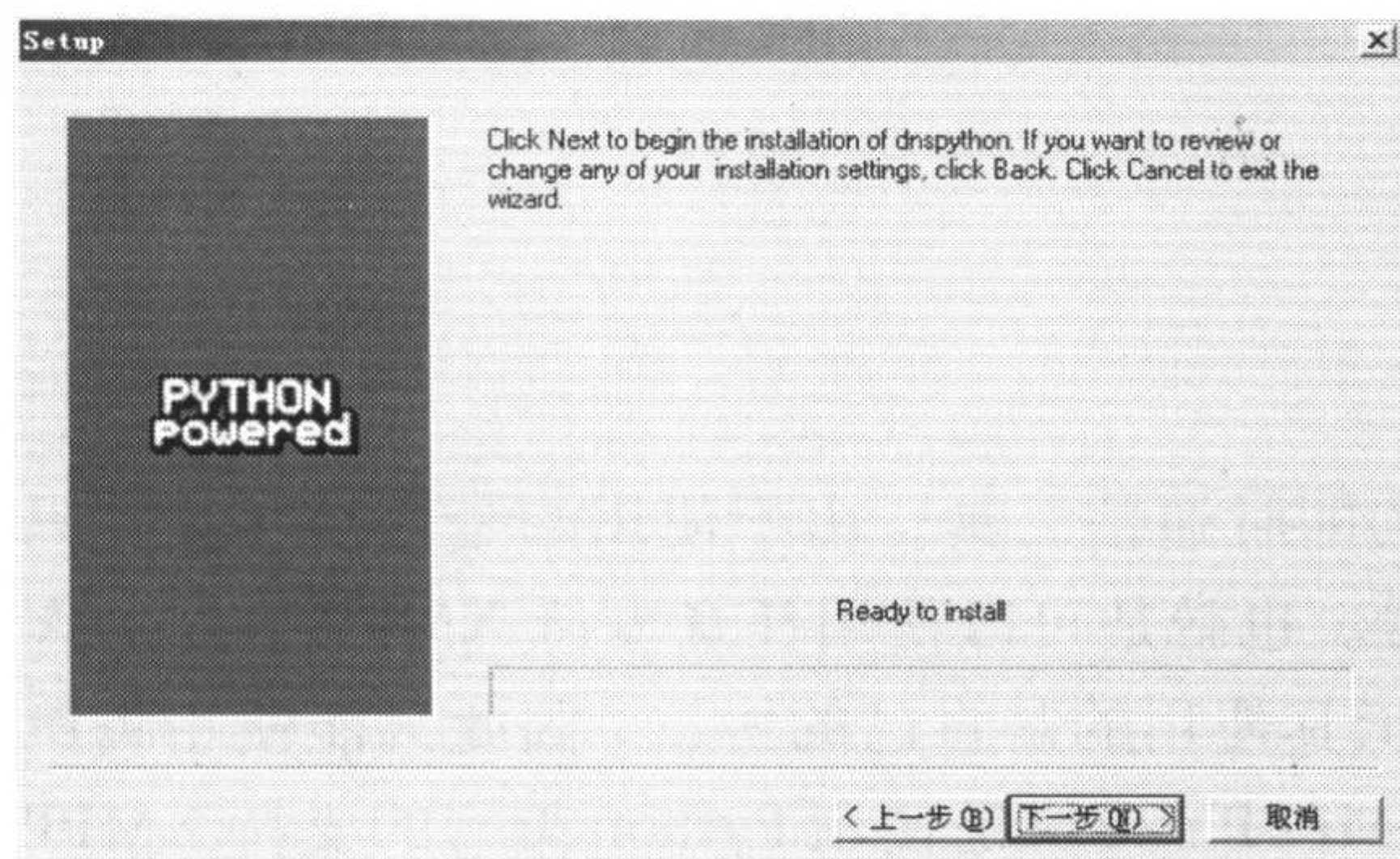


图 18-8 确认安装

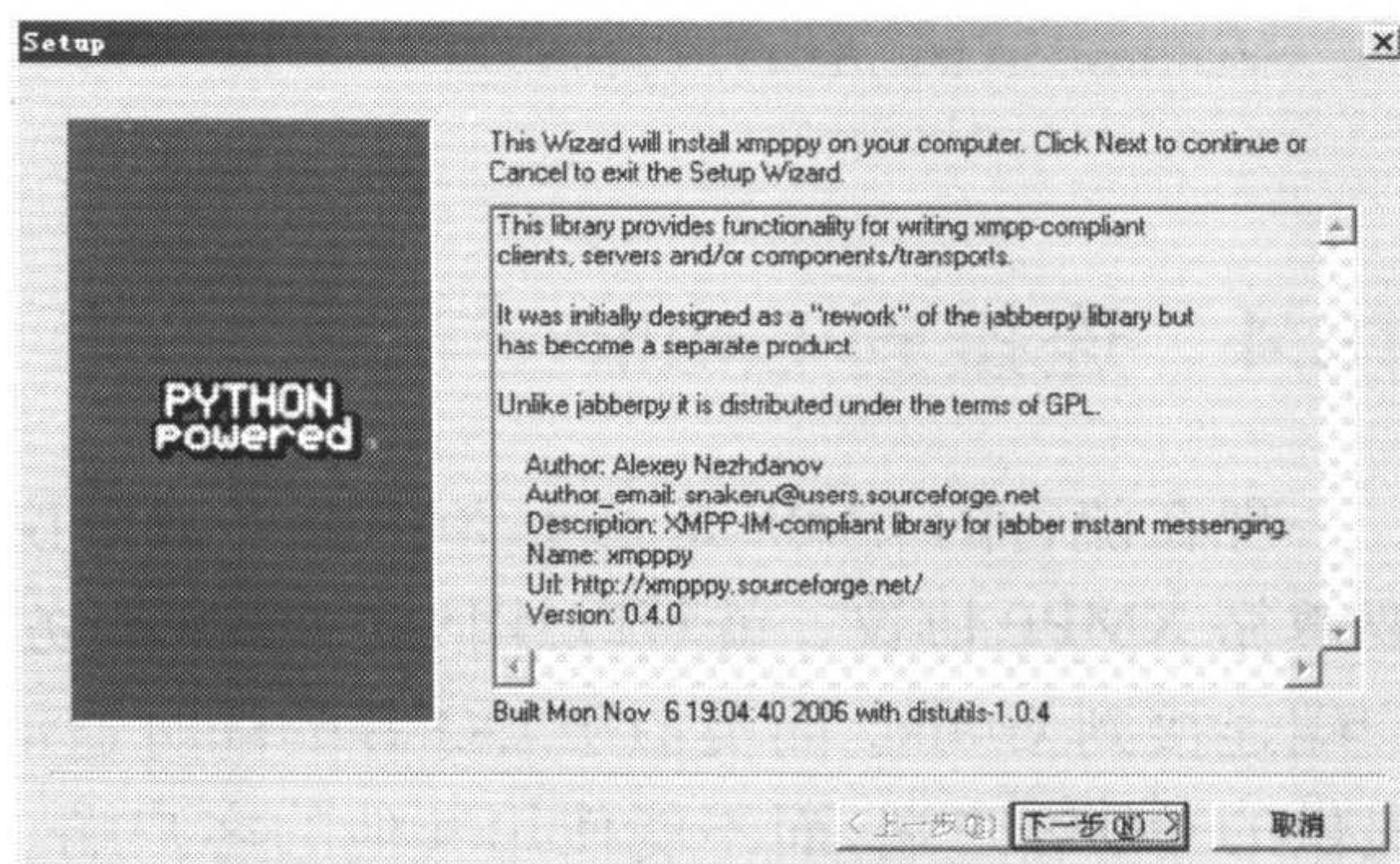


图 18-9 XMPPPY 安装界面

(2) 单击【下一步】按钮，进入安装选择路径界面，如图 18-10 所示。根据要使用的 Python 版本，选择安装路径。

(3) 单击【下一步】按钮，进入确认安装界面，如图 18-11 所示。单击【下一步】按钮，可以完成 XMPPY 的安装。

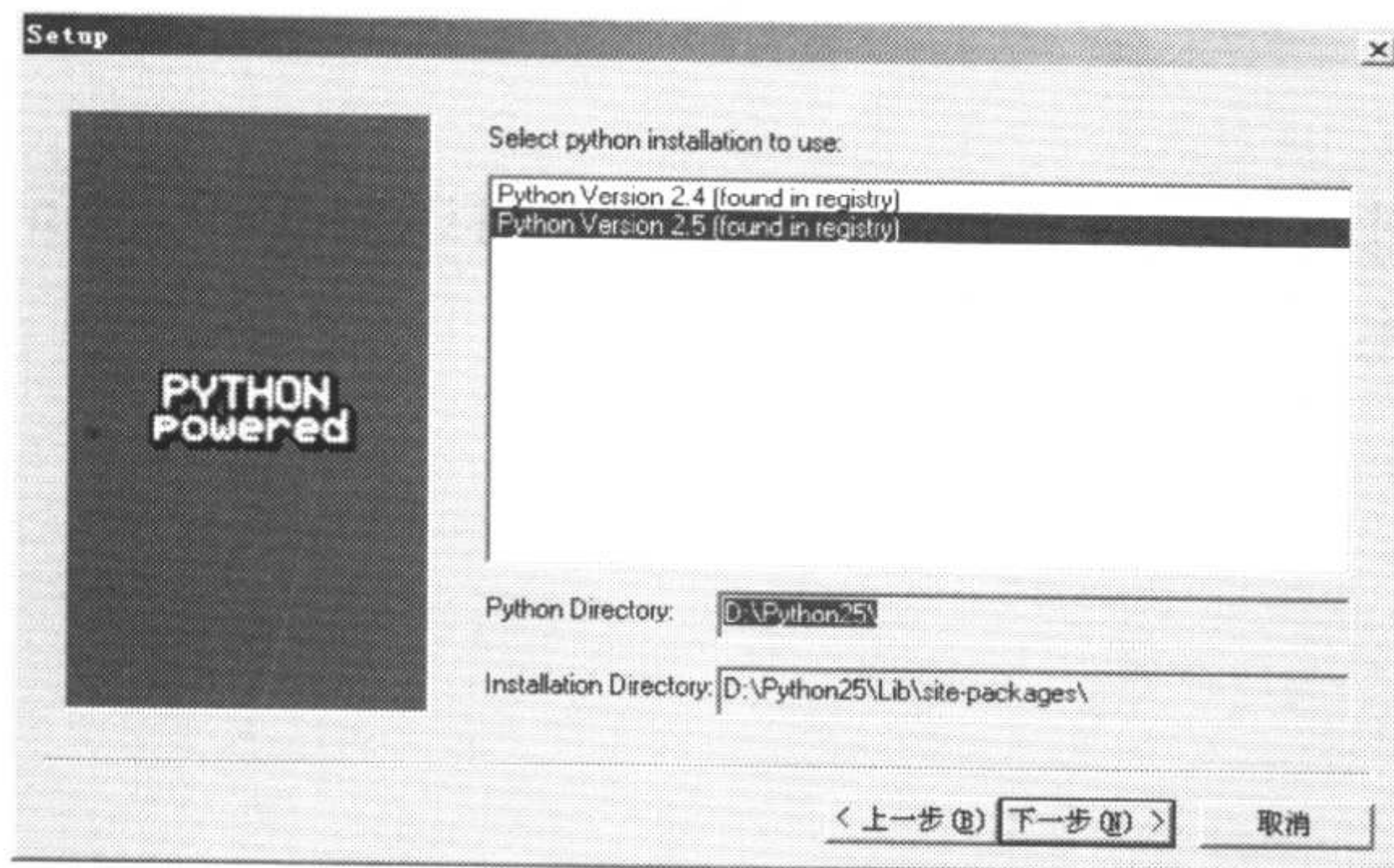


图 18-10 选择安装路径

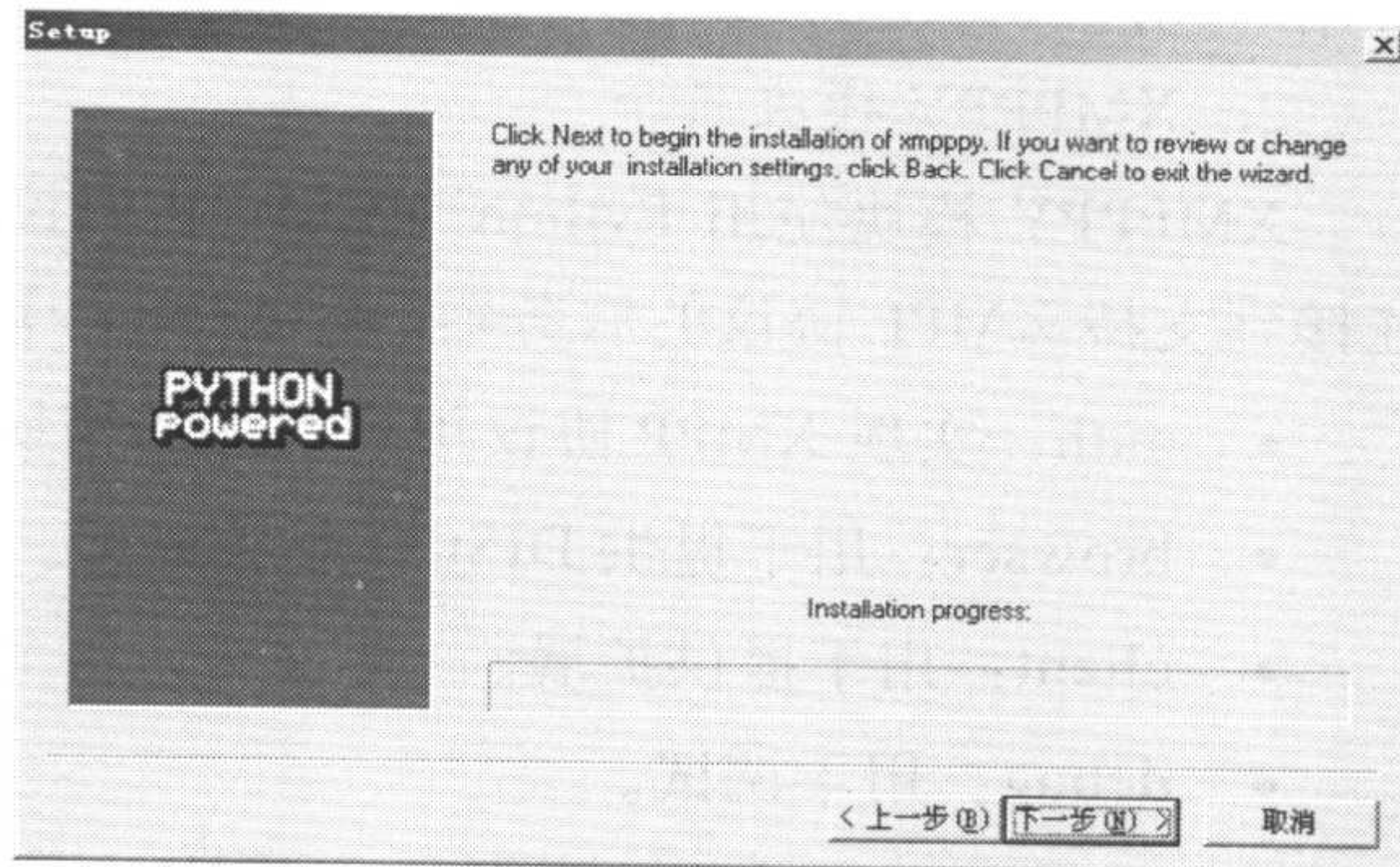


图 18-11 确认安装

18.4.2 使用 XMPPY

XMPPY 模块是对 XMPP 协议的封装。XMPP 是基于可扩展标记语言 (XML) 的协议，它用于即时消息。使用 XMPPY 模块可以在 Python 中使用 XMPP 协议。

1. XMPP 协议简介

XMPP (Extensible Messaging and Presence Protocol)，即可扩展消息处理现场协议。XMPP 是基于 XML (可扩展标记语言) 的协议。使用 XMPP 协议可以向因特网上的其他任何人发送即时消息。XMPP 不依赖于操作系统和浏览器。由于 XMPP 是一种基于 XML 的协议，它继承了 XML 的灵活性。因此，基于 XMPP 协议的应用具有很强的扩展性。XMPP 的基本网络架构是由客户端、服务器、网关 3 部分组成的。各部分描述如下所示。

- 客户端 客户端通过 TCP 连接直接连接到服务器，一般来说服务器的默认连接端口为 5222。客户端连接到服务器后，就可以使用 XMPP 协议进行通信。
- 服务器 服务器主要用于管理连接会话，存储用户信息等。由于 XMPP 协议的开放性，还可以根据具体情况在服务器实现所需要的功能包。
- 网关 网关主要用于与非 XMPP 协议的即时通信系统进行交互，转换协议。多数情况下并不需要网关。

Google 推出的即时通信软件 Gtalk 就采用了 XMPP 协议，这意味着只要支持 XMPP 协议的客户端都可以连接到 Gtalk，用户也可以自己编写功能更强大的客户端。尽管 Gtalk 只有 Windows 平台下的版本，但是有很多 Linux 下的即时通信客户端支持 XMPP 协议，如 GAIM。因此，完全可以使用其他的客户端代替 Gtalk，在不同的系统平台下使用。而目前国内十分

流行的 QQ 则使用封闭的通信协议，只能使用 QQ 官方提供的客户端。

由于 Gtalk 的开放性，很多用户都开始选择 Gtalk 作为即时通信工具。现在网上流行的基于 Python 的 Gtalk 聊天机器人。使用 Gtalk 的聊天机器人可以建立和 QQ 群一样的群组，使得用户可以在一起进行讨论。由于 XMPP 的开放性，使得 Gtalk 的聊天机器人可以实现很多实用的功能。开放的 Gtalk 将会更加流行。

2. XMPPY 模块简介

XMPPY 模块使用 Python 对 XMPP 协议进行了封装，使用 XMPPY 可以使用 Python 连接到支持 XMPP 协议的服务器。XMPPY 中主要有以下几个模块。

- auth: 实现 XMPP 协议用户认证的模块。
- browser: 用于提供 DISCO 服务框架。
- client: 用于提供扩展。
- debug: 用于调试。
- dispatcher: XMPPY 的主要模块。
- features: 包含一些不能分割的内容。
- filetransfer: 用于实现 JEP-0047。
- protocol: 实现 XMPP 通信协议模块。
- roster: 用于实现用户列表。
- simplexml: 用于处理 XML。
- transports: 底层传输协议。

XMPPY 模块较为庞大，详细的内容可以参考 XMPPY 的帮助文档，这里不再赘述。

3. 使用 XMPPY 连接 Gtalk

如下所示的 gtalk.py 使用 XMPPY 模块创建了一个简单的 Gtalk 客户端。为了简便起见，在 gtlak.py 脚本中创建连接后就进入循环，当有用户发送消息，就输出消息内容和用户名，然后向用户发送消息。

```
# -*- coding:utf-8 -*-
# file: gtalk.py
#
import xmpp
def GetMessage(client, message):
    text = message.getBody()
    people = message.getFrom()
    print 'GET: %s FROM: %s' % (text, people)
    client.send(xmpp.protocol.Message(people,
        'GET: ' + text, typ="chat"))
user = raw_input('User:')
password = raw_input('Password:')
jid = xmpp.protocol.JID(user + '@gmail.com')
```

导入模块
消息处理函数
获得消息内容
获得发信者
打印消息
发送消息

获取用户名
获取密码
创建 JID

```
client = xmpp.Client(jid.getDomain(), debug=[])           # 创建客户端
client.connect(('talk.google.com', 5222))                # 连接服务器
client.auth(user, password)                              # 用户认证
Roster = client.getRoster()                              # 获取用户列表
names = Roster.getItems()                                # 获取用户名
for name in names:                                       # 循环输出用户
    status = Roster.getStatus(name)
    print name,
    print status
client.RegisterHandler('message', GetMessage)           # 注册消息回调函数
client.sendInitPresence()                                # 设置在线
while 1:                                                  # 进入循环
    try:
        client.Process(1)
    except KeyboardInterrupt:                             # 处理 Ctrl+c
        break
```

运行 gtalk.py 脚本后会出现如下的提示。

```
An error occurred while looking up _xmpp-client._tcp.talk.google.com
```

该提示并不影响脚本运行。另外，如果处于局域网中，可能无法连接到 Gtalk 的服务器。

第 19 章 处理 HTML 与 XML

在前面章节中已经提到 HTML 和 XML。HTML 主要用于 Web，虽然在服务器端可以使用多种技术，但在客户端，多数情况下都是使用 HTML。而 XML 的用途则较为广泛。XML 可以用于创建协议，例如 XMPP 协议；用于创建文件格式，例如 wxPython 中使用的资源文件。在 Python 中可以使用标准的模块对 HTML 和 XML 进行处理。

19.1 处理 HTML

在 Python 中可以使用 HTMLParser 模块处理 HTML，获取页面中感兴趣的内容。HTMLParser 模块提供了对 HTML 标记处理的方法。如果有些内容不能使用 HTMLParser 处理，还可以自己编写正则表达式进行匹配。

19.1.1 HTMLParser 模块简介

在使用 HTMLParser 模块处理 HTML 时，首先应继承 HTMLParser 模块中的 HTMLParser 类，然后重载相关的处理方法。使用 HTMLParser 对象的 feed 方法可以向 HTMLParser 传递数据。其原型如下所示。

```
feed(data)
```

其参数如下。

- data: 传递的数据。

当向 HTMLParser 对象传递数据后，其就开始对数据进行处理。使用 HTMLParser 对象的 close 方法，可以强制处理 feed 方法存在在缓冲区中的数据。使用 HTMLParser 对象的 reset 方法可以重新设置对象实例，进行新一轮的数据处理。使用 HTMLParser 对象的 getpos 方法可以获得当前处理的行号和偏移位置。

在使用 HTMLParser 处理 HTML 的过程中，遇到某些标记或者数据就会调用相应的方法。一般情况下，在脚本中需要重载这些方法，以完成对 HTML 的处理。当 HTMLParser 每次遇到一个起始标记时，会调用 handle_starttag 方法。其原型如下所示。

```
handle_starttag(tag, attrs)
```

其参数含义如下。

- tag: HTMLParser 遇到的标记。
- attrs: 标记的属性。

当 HTMLParser 遇到类似于
的标记时, 将调用 handle_startendtag 方法。其原型如下所示。

```
handle_startendtag(tag, attrs)
```

其参数含义如下。

- tag: HTMLParser 遇到的标记。
- attrs: 标记的属性。

当 HTMLParser 遇到结束标记时, 会调用 handle_endtag 方法。其原型如下所示。

```
handle_endtag(tag)
```

其参数含义如下。

- tag: HTMLParser 遇到的结束标记。

使用 HTMLParser 的 handle_data 方法可以处理标记间的数据。其原型如下所示。

```
handle_data(data)
```

其参数含义如下。

- data: 标记间的数据。

当 HTMLParser 遇到 HTML 中的注释时, 将调用 handle_comment 方法。其原型如下所示。

```
handle_comment(data)
```

其参数含义如下。

- data: 注释内容。

除了 HTMLParser 模块中的 HTMLParser 类以外, 在 htmllib 模块中也提供了一个简单的 HTMLParser 类。其提供了处理超链接和图片的方法。htmllib 模块中 HTMLParser 类处理超链接的方法原型如下所示。

```
anchor_bgn(href, name, type)
anchor_end()
```

其参数如下。

- href: href 属性。
- name: name 属性。
- type: type 属性

htmllib 模块中 HTMLParser 类处理图片的方法原型如下所示。

```
handle_image(source, alt, ismap, align, width, height)
```

其参数含义如下。

- source: source 属性。

- alt: alt 属性。
- ismap: ismap 属性。
- align: align 属性。
- width: width 属性。
- height: height 属性。

19.1.2 获取页面图片地址

在 Python 的 HTMLParser 模块中提供了处理标记的方法，由于在网页中图片都是以“”标记嵌入到网页中的，因此要获取页面中图片的地址只要处理“”标记即可。如下所示的 GetImage.py 脚本重载了 HTMLParser 类的 handle_starttag 方法，对“”标记进行处理分别获得 GIF 和 JPG 图片的地址。由于没有对地址做进一步的处理，因此脚本获得的可能是图片的相对地址。

```
# -*- coding:utf-8 -*-
# file: GetImage.py
#
import Tkinter
import urllib
import HTMLParser

class MyHTMLParser(HTMLParser.HTMLParser):
    def __init__(self):
        HTMLParser.HTMLParser.__init__(self)
        self.gifs = []
        self.jpgs = []
    def handle_starttag(self, tags, attrs):
        if tags == 'img':
            for attr in attrs:
                for t in attr:
                    if 'gif' in t:
                        self.gifs.append(t)
                    elif 'jpg' in t:
                        self.jpgs.append(t)
                    else:
                        pass
    def get_gifs(self):
        return self.gifs
    def get_jpgs(self):
        return self.jpgs

class Window:
    def __init__(self, root):
        self.root = root
        self.label = Tkinter.Label(root, text = '输入 URL:')
        self.label.place(x = 5, y = 15)
        self.entryUrl = Tkinter.Entry(root,width = 30)
        self.entryUrl.place(x = 65, y = 15)
```

```
# 创建 HTML 解析类
# 创建列表，保存 gif
# 创建列表，保存 jpg
# 处理起始标记
# 处理图片
# 添加到 gif 列表
# 添加到 jpg 列表
# 返回 gif 列表
# 返回 jpg 列表
# 创建组件
```

```

self.get = Tkinter.Button(root,
    text = '获取图片', command = self.Get)
self.get.place(x = 280, y = 15)
self.edit = Tkinter.Text(root,width = 470,height = 600)
self.edit.place(y = 50)
def Get(self):
    url = self.entryUrl.get()
    page = urllib.urlopen(url)
    data = page.read()
    parser = MyHTMLParser()
    parser.feed(data)
    self.edit.insert(Tkinter.END, '====GIF====\n')
    gifs = parser.get_gifs()
    for gif in gifs:
        self.edit.insert(Tkinter.END, gif + '\n')
    self.edit.insert(Tkinter.END, '====\n')
    self.edit.insert(Tkinter.END, '====JPG====\n')
    jpgs = parser.get_jpgs()
    for jpg in jpgs:
        self.edit.insert(Tkinter.END, jpg + '\n')
    self.edit.insert(Tkinter.END, '====\n')
    page.close()
root = Tkinter.Tk()
window = Window(root)
root.minsize(600,480)
root.mainloop()

```

获取 URL
打开 URL
读取 URL 内容
生成实例对象
处理 HTML 数据
输出数据

运行 GetImage.py 脚本后，在文本框中输入网址，单击【获取图片】按钮，将对网址进行处理，获取网页中的图片地址，如图 19-1 所示。由于很多网站使用的 HTML 不是标准的语法格式，因此，使用 HTMLParser 处理时，并不一定能够获得所有的图片。

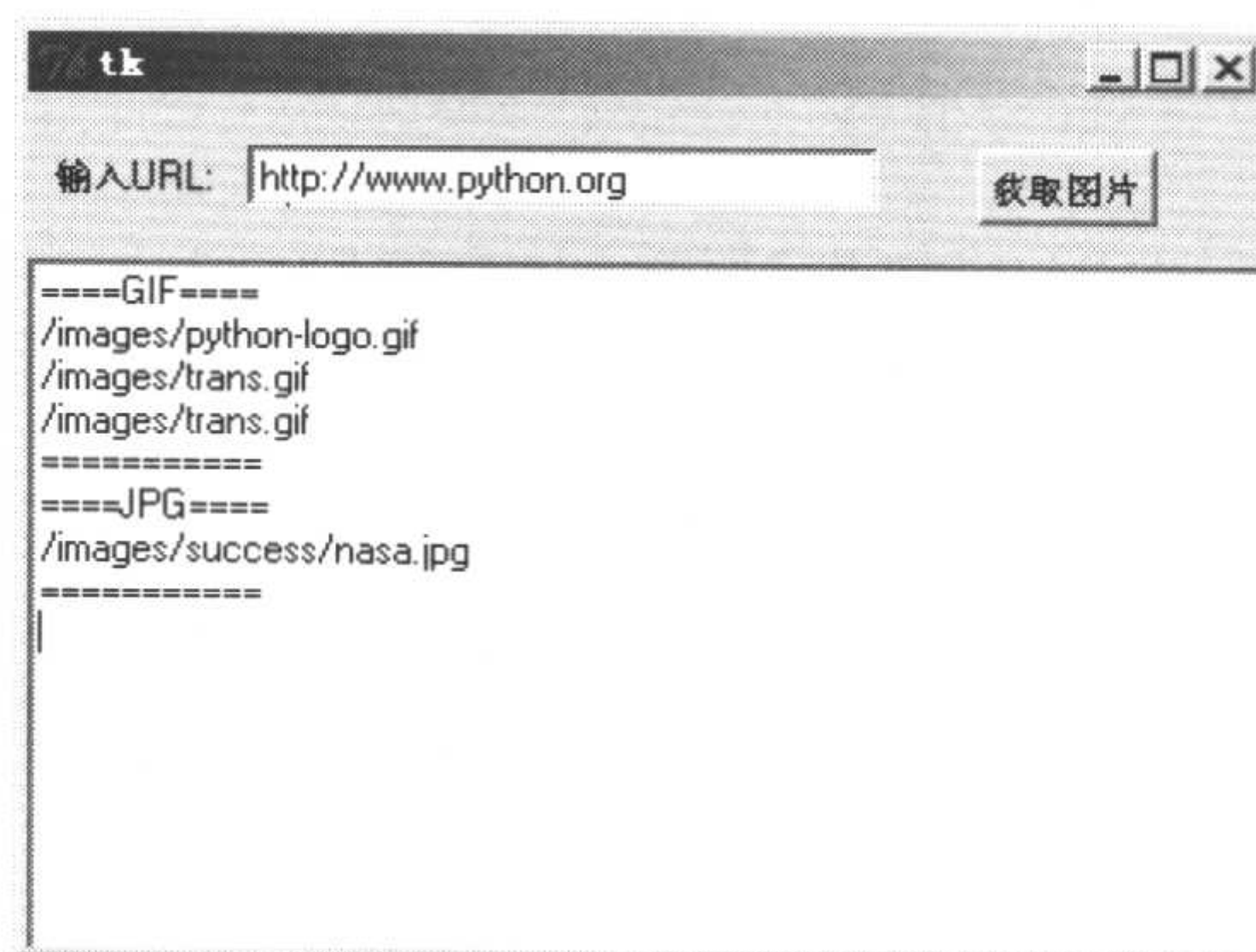


图 19-1 获取网站图片地址

19.1.3 查看天气预报

有很多网站提供了免费的天气预报服务，但是如果每次都要登录网站查看天气，则有些

麻烦。使用 Python 完全可以编写一个脚本，自动获取提供免费天气预报网站的内容，获得所需要的天气信息。

以中文雅虎的免费天气预报服务为例 <http://weather.cn.yahoo.com>，通过访问网站查找城市天气预报可以发现，其天气显示地址形式如下所示。

`http://weather.cn.yahoo.com/weather.html?city=%E5%AE%9C%E6%98%8C`

其中，city 后的为经过编码后的城市名。因此，如果要获取某一城市的天气，只需修改 city 后的城市名编码即可。通过查看页面源代码可以发现天气处于如下代码中。

```
<div class="yui-m">
  <!--{{start:detail -->
<div id="map" class="map">
  <div class="pd">
    <h3>您现在看到的天气由中央气象台今日 16:00 时发布</h3>
    <div class="cnt">
      <div class="tl"></div>
      <div class="tr"></div>
      <div class="bl"></div>
      <div class="br"></div>
      <div id="map">
        <div class="map_cnt2">
          <div class="dt_a">
            <div class="l" id="fck">宜昌</div>
            <div class="r">
              <!--<a href="javascript:void(0)" onclick="wck2()">设置为默认城市</a>-->
            </div>
          <div class="x"></div>
          <div class="dt_b">
            <div class="l1">
              <!--today -->
            <div class="dt_c">
              <div class="tn">2007-5-16</div>
            <p>
              <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#versi
on=7,0,19,0" width="64" height="64">
                <param name="movie" value="http://cn.yimg.com/i/wea/v1/ico/w3.swf" />
                <param name="quality" value="high" />
                <embed src="http://cn.yimg.com/i/wea/v1/ico/w3.swf" quality="high" pluginspage=
"http://www.macromedia.com/go/getflashplayer" type="application/x-shockwave-flash"
width="64" height="64"></embed>
              </object>
            </p>
            <div class="dt_d"> <span class="ft1">多云转晴</span><br />
              <span class="tmp"><span class="htp">29 °C </span> ~ <span class="lotp">18 °C
</span></span><br />
              微风</div>
            </div>
            <!--//today -->

```

```

</div>
<div class="l2">
<!--today -->
<div class="dt_c">
<div class="tn">2007-5-17</div>
<p>
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=7,0,19
,0" width="64" height="64">
<param name="movie" value="http://cn.yimg.com/i/wea/v1/ico/w3.swf" />
<param name="quality" value="high" />
<embed src="http://cn.yimg.com/i/wea/v1/ico/w3.swf" quality="high" pluginspage=
"http://www.macromedia.com/go/getflashplayer" type="application/x-shockwave-flash"
width="64" height="64"></embed>
</object>
</p>
<div class="dt_d"> <span class="ft1">多云转晴</span><br />
<span class="tmp"><span class="hitp">32 °C </span> ~ <span class="lotp">19 °C
</span></span><br />
微风</div>
</div>
<!--//today -->
</div>
<div class="l3">
<!--today -->
<div class="dt_c">
<div class="tn">2007-5-18</div>
<p>
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=7,0,19
,0" width="64" height="64">
<param name="movie" value="http://cn.yimg.com/i/wea/v1/ico/w1.swf" />
<param name="quality" value="high" />
<embed src="http://cn.yimg.com/i/wea/v1/ico/w1.swf" quality="high" pluginspage=
"http://www.macromedia.com/go/getflashplayer" type="application/x-shockwave-flash"
width="64" height="64"></embed>
</object>
</p>
<div class="dt_d"> <span class="ft1">晴</span><br />
<span class="tmp"><span class="hitp">33 °C </span> ~ <span class="lotp">20 °C
</span></span><br />
微风</div>
</div>
<!--//today -->

```

可以看到上述的代码非常复杂，主要使用了“<div>”标记。而在HTMLParser中对“<div>”标记的支持并不是很好，通过编写一个测试脚本可以发现，并不能很好地处理该页面。既然不能通过HTMLParser处理，只好使用正则表达式。实际上HTMLParser也是使用的正则表达式。

通过分析上述代码可以发现，最主要的天气信息是处于<div class="dt_d">和</div>标记

之间。网址显示的是三天的天气情况,可以看到每天天气的主要信息都处于<div class="dt_d">和</div>标记之间。而在页面的其他位置并没有再出现<div class="dt_d">标记,这样可以通过编译如下的正则表达式匹配<div class="dt_d">和</div>标记之间的内容。

```
re.compile(r'(?<=dt_d...)+?(?=</div)',re.S|re.I|re.U)
```

为了检验匹配的结果,可以编写一个简单的测试脚本,这里就不给出代码了。正确的匹配结果应该是如下所示的代码。

```
<span class="ft1">多云转晴</span><br />
<span class="tmp"><span class="hitp">29 °C </span> ~ <span class="lotp">18 °C
</span></span><br />
微风
```

可以看到主要的天气信息都已经获取了。接下来分别匹配天气情况、最高温度和最低温度等。不难发现,它们也都处于不同的标记之间。对于天气情况,可以编写如下正则表达式进行匹配。

```
weather = re.compile(r'(?<=ft1...)+?(?=</span)',re.S|re.I|re.U)
```

对于最高温度,可以编写如下正则表达式进行匹配。

```
hitp = re.compile(r'(?<=hitp...)+?(?=</span)',re.S|re.I|re.U)
```

对于最低温度,可以编写如下正则表达式进行匹配。

```
lotp = re.compile(r'(?<=lotp...)+?(?=</span)',re.S|re.I|re.U)
```

如下所示的 GetWeather.py 脚本,主要使用上述分析所得的正则表达式对页面进行处理,获取天气情况。

```
# -*- coding:utf-8 -*-
# file: GetWeather.py
#
import urllib
import re
import Tkinter
import datetime
class Window:
    def __init__(self, root):
        self.root = root
        self.label = Tkinter.Label(root, text = '输入城市:')
        self.label.place(x = 5, y = 15)
        self.entryCity = Tkinter.Entry(root)
        self.entryCity.place(x = 65, y = 15)
        self.get = Tkinter.Button(root,
            text = '获取天气', command = self.Get)
        self.get.place(x = 230, y = 15)
        self.edit = Tkinter.Text(root,width = 200,height = 150)
        self.edit.place(y = 50)
    def Get(self):
        city = self.entryCity.get().encode('utf-8')
        addr = 'http://weather.cn.yahoo.com/weather.html?city=%s'
        cityencode = urllib.quote(city)
        data = urllib.urlopen(addr % cityencode)
```

创建组件

获取城市
查看天气地址
对中文进行编码
打开网页

第19章 处理HTML与XML

```

s = data.read() # 获取网页数据
m = re.compile(r'(?<=dt_d...)+?(?=</div)', re.S|re.I|re.U) # 编译正则表达式

weather = re.compile(r'(?<=ft1...)+?(?=</span)', re.S|re.I|re.U)
hitp = re.compile(r'(?<=hitp...)+?(?=</span)', re.S|re.I|re.U)
lotp = re.compile(r'(?<=lotp...)+?(?=</span)', re.S|re.I|re.U)
today = datetime.date.today() # 获得当前日期
year = today.strftime('%Y') # 日期格式化
month = today.strftime('%m')
day = today.strftime('%d')
days = ['今天', '明天', '后天']
a = 0
for i in m.findall(s): # 循环处理天气
    d = str(int(day)+a)
    dates = '%s-%s-%s' % (year, month, d)
    self.edit.insert(Tkinter.END, # 输出日期和城市
        '%s%s(%s)\n' % (city, days[a], dates))
    a = a + 1
    for j in weather.findall(i):
        self.edit.insert(Tkinter.END, j + '\n') # 输出天气
    for k in hitp.findall(i):
        self.edit.insert(Tkinter.END, '最高温度: ')
        self.edit.insert(Tkinter.END, k + '\n') # 输出最高温度
    for l in lotp.findall(i):
        self.edit.insert(Tkinter.END, '最低温度: ') # 输出最低温度
        self.edit.insert(Tkinter.END, l + '\n')
    self.edit.insert(Tkinter.END,
        '*****\n')

data.close()
root = Tkinter.Tk()
window = Window(root)
root.minsize(300, 245)
root.mainloop()

```

运行 GetWeather.py 后, 在文本框中输入城市名, 单击【获取天气】按钮, 即可在多行文本框中显示 3 天的天气信息, 如图 19-2 所示。

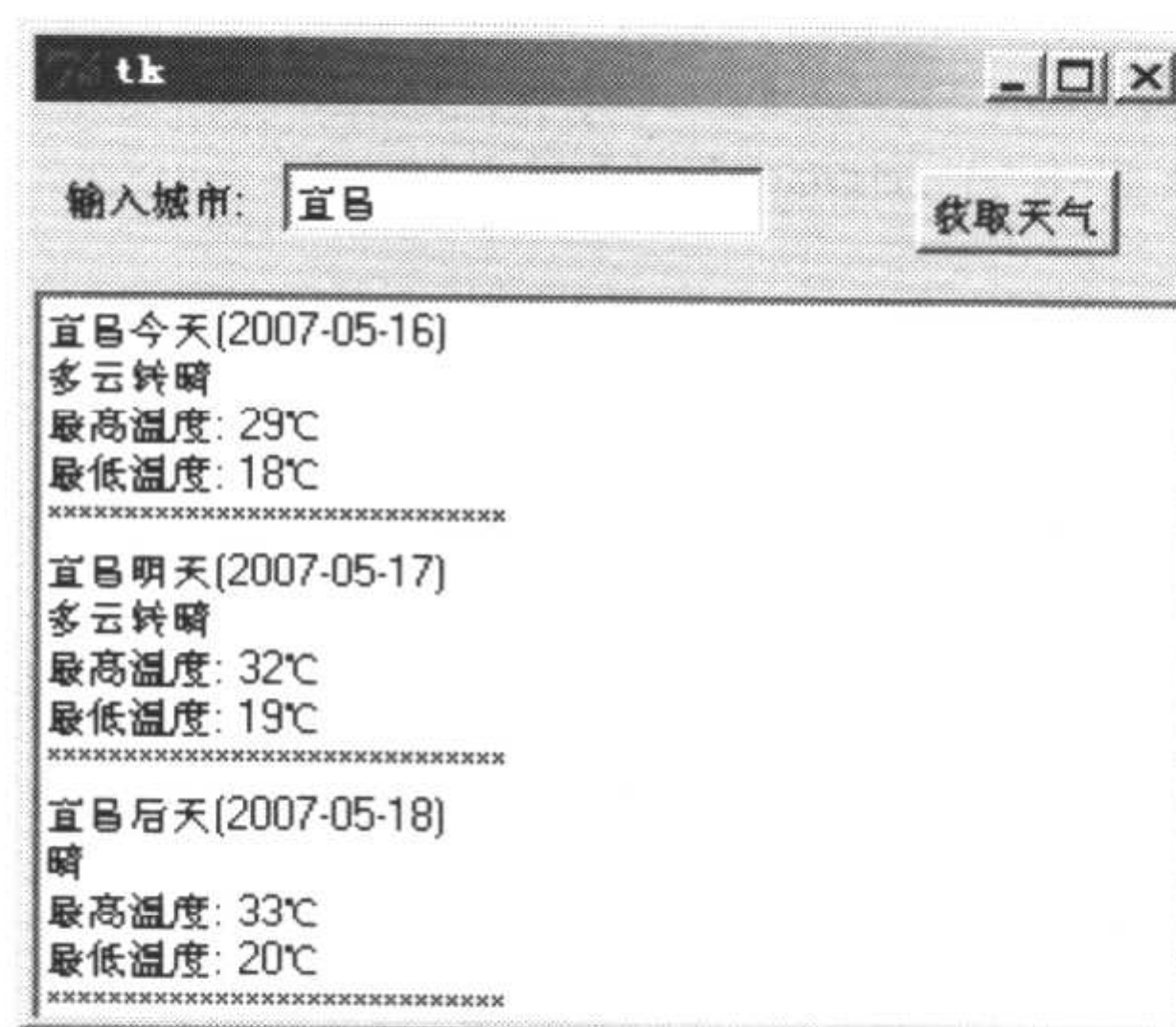


图 19-2 获取天气

在第 10 章中编写了一个修改 IE 标题栏的 SetIE.py 脚本, 可以将 GetWeather.py 脚本中获取天气预报的部分添加到 SetIE.py 脚本中, 将当天的天气预报添加到 IE 标题栏中。修改后的 SetIE.py 脚本如下所示。

```
# -*- coding: utf-8 -*-
# file: SetIE.py
#
import datetime
import string
import win32api
import win32con
import urllib
import re
# 要修改的注册表项
keyname = 'Software\Microsoft\Internet Explorer\Main'
# 要设置为主页的网址
page = 'www.python.org'
# 获取当前日期
today = datetime.date.today()
# 将日期格式化为 xxxx 年 xx 月 xx 日的形式
title = today.strftime('%Y')+'年'+today.strftime('%m')+'月'+today.strftime('%d')+'日'
# 天气预报地址, 需根据需要修改
addr = 'http://weather.cn.yahoo.com/weather.html?city=%E5%AE%9C%E6%98%8C'
# 获取网页, 并处理数据
data = urllib.urlopen(addr)
s = data.read()
m = re.compile(r'(?<=dt_d...)+?(?<=

```

```
if StartPage[0] != page:
    win32api.RegSetValueEx(key, 'Start Page', 0, win32con.REG_SZ, page)
# 设置 IE 的标题栏为 xxxx 年 xx 月 xx 日
win32api.RegSetValueEx(key, 'Window Title', 0, win32con.REG_SZ, title)
# 关闭注册表
win32api.RegCloseKey(key)
```

运行 SetIE.py 后, 打开 IE, 可以看到 IE 标题中已经包含了天气信息, 如图 19-3 所示。



图 19-3 向 IE 标题栏中添加天气信息

19.2 处理 XML

相对于 HTML 而言, XML 具有更好的结构和语法, XML 文件更加清晰。HTML 不具备扩展性, 而且标准也不统一, 因此在上一节中处理天气时, 只能使用正则表达式进行处理。处理 XML 则相对要容易得多, 而且 XML 具有很强的扩展性。

在前面的章节中已经多次接触到 XML。XML 作为具有很强扩展性的标记语言应用得十分广泛。XML 不具有自己的标记, 使用 XML 时需要自己建立标记, 但这并不影响 XML 的结构化。另外, XML 区分大小写。

19.2.1 XML 基础

一般来说, XML 文档可能包含以下几部分内容。

1. XML 声明

XML 声明中包含三部分内容: version、encoding 和 standalone。其中 version 用于指定所使用的 XML 规范的版本号。encoding 用于指定 XML 文档所使用的编码格式。standalone 用于指定文档是否独立。如下所示为几种不同的 XML 声明。


```
<?xml version="1.0"?>
<?xml version="1.0" encoding="utf-8"?>
<?xml version='1.0' encoding='utf-8' standalone='no'?>
```

2. 根元素

XML 文档都具有一个根元素。根元素是 XML 文档中的第一个元素。在 XML 文档的根元素中包含了其他的元素。

3. 元素和属性

元素是处于“<>”中的标记,而元素属性是标记后的附加信息。如下所示的 XML 中 image 为元素, width 和 height 为属性,“=”后以单引号包围的为属性值。属性值既可以使用单引号包围,也可以使用双引号。XML 中的标记是成对出现的,如果只有一个标记,则可以写成“<one/>”的形式。

```
<image width='640' height='480'></image>
```

4. 字符数据

字符数据是处于标记间的数据。XML 文档中的主要内容是字符数据。如下所示的 XML,其字符数据为“Alien”。

```
<name>
Alien
</name>
```

5. CDATA 块

CDATA 块使用“<![CDATA[”和“]]>”包围内容。CDATA 块适用于大段的文本,CDATA 块中可以包含特殊的字符。如果不使用 CDATA 块,特殊字符如“<”、“>”等,则应该写成实体引用的形式“<”、“>”。

6. 注释

注释是以“<!--”和“-->”包围的区间。在注释内容中不允许出现“--”。

7. 处理指令

处理指令以“<?”和“?>”包围数据。使用处理指令可以向处理器传递指令,使其按特定的方式处理 XML 文档中的数据。如下所示的 XML 文档是第 13 章中 wxPython 的资源文件。

```
<?xml version="1.0" encoding="utf-8"?>
<resource>
  <object class="wxFrame" name="frame">
    <title>wxPython XRC</title>
    <object class="wxPanel" name="panel">
      <object class="wxStaticText" name="label">
        <label>Input</label>
        <pos>50,70</pos>
      </object>
      <object class="wxTextCtrl" name="text">
        <pos>120,70</pos>
```

```

    </object>
    <object class="wxButton" name="button">
        <label>Ok</label>
        <pos>100,120</pos>
    </object>
</object>
<size>300,200</size>
</object>
</resource>

```

该 XML 文档的声明部分如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
```

其根元素为“resource”，其中组件都是处于“object”标记之间。“object”标记的“class”属性和“name”属性指明了组件的类和组件名。组件的位置信息是通过“pos”标记间的字符数据表示的。上述 XML 文档没有包含注释 CDATA 块。可以看出使用 XML 文档具有很清晰的结构，通过标记可以很容易地明白其表示的含义。

19.2.2 文档类型定义

DTD (Document Type Definition)，即文档类型定义，主要用于描述 XML 文档包含的内容和 XML 文档的布局结构。使用 DTD 可以验证 XML 文档结构的正确性。在 XML 文档中，可以使用 XML 文档头，用 URL 地址指定 XML 文档所使用的 DTD，如下所示。

```
<!DOCTYPE example SYSTEM "http://www.w3c.com/dtd/portal.dtd">
```

除此以外，DTD 还可以直接包含在 XML 文档中。通过阅读 DTD 即可知道 XML 文件中元素的安排以及元素所具有的属性等。

DTD 的语法很简单，在 DTD 中包含元素的声明和元素的属性定义。其中元素的声明形式如下所示。

```
<!ELEMENT 元素名 元素内容模型>
```

元素内容模型可以使用以下几种形式。

1. #PCDATA

元素可能包含字符数据、XML 元素中的普通文本。其使用形式如下所示。

```
<!ELEMENT element-name (#PCDATA)>
```

2. EMPTY

空元素，不包含内容，写成<tag/>的形式。其使用形式如下所示。

```
<!ELEMENT image EMPTY>
```

ANY， 可以包含任何元素或者字符数据。其使用形式如下所示。

```
<!ELEMENT image ANY>
```

3. 子元素

元素所包含的子元素，可以使用类似于正则表达式中的元字符来表示子元素的数目或者顺序等。可以使用的元字符如表 19-1 所示。

表 19-1 XML 中的元字符

符 号	描 述	符 号	描 述
*	零个或多个元素	,	元素按给定的顺序出现
+	一个或多个元素		包含的元素可替换
?	零个或一个元素	()	组合元素

如下所示的元素定义表示元素包含零个或一个 A 元素。

```
<!ELEMENT NAME A+>
```

如下所示的元素定义表示元素包含 A、B 两个元素，其 A 元素处于 B 元素之前。

```
<!ELEMENT NAME (A,B)>
```

一般来说，元素的属性定义应紧跟在元素声明之后，但不是必须的。也可以将元素属性定义和元素定义分成两部分。元素属性定义形式如下所示。

```
<!ATTLIST 元素名
  属性名 1 属性数据类型 属性行为
  属性名 2 属性数据类型 属性行为
  属性名 3 属性数据类型 属性行为
  .....
>
```

属性数据类型可以是以下几种。

- CDATA：任意的文本字符串，可以包含空格等。
- NMTOKEN：第一个字母必须是字母、表意字符或下划线的字符串。
- NMTOKENS：和 NMTOKEN 类似，但是字符中应包含空格。
- ID：表示该值在 XML 文档中唯一。
- IDREF：指向 XML 文档中另一元素的 ID。
- IDREFS：和 NMTOKENS 类似，包含用空格分隔的多个 ID 引用。
- ENTITY：实体名。
- 枚举类型：列举出属性所有可能的值，不同的值之间用“|”分隔。

属性的行为类型如表 19-2 所示。

表 19-2 属性行为

属 性 行 为	描 述	属 性 行 为	描 述
“default”	默认值	#REQUIRED	属性为必需的
#IMPLIED	属性为可选的	#FIXED	属性值为固定的

19.2.3 命名空间

XML 的命名空间是为了避免元素命名和属性名的冲突。使用 XML 的命名空间可以指定 XML 元素名和属性名的使用范围，这样就可以在一定程度上避免名字冲突。使用命名空间的

元素名或属性名都是由两部分组成的，第一部分是命名空间前缀，第二部分是元素名或者属性名。这两部分由“:”分隔。

命名空间的声明可以使用直接定义和默认定义两种方式。其中直接定义命名空间的形式如下所示。

xmlns: 命名空间前缀=命名空间名

默认定义命名空间的形式如下所示。

xmlns=命名空间名

使用默认的命名空间定义，可以省略命名空间前缀部分。如下所示的 XML 使用了命名空间。

```
<ctgu:menmber xmlns:ctgu = "http://www.example.org/example.dta">
  <ctgu:item>
    <ctgu:name>Tom</ctgu:name>
    <ctgu:ID>2002105101</ctgu:ID>
    <ctgu:Phone>6390001</ctgu:Phone>
    <ctgu:Zip>443002</ctgu:Zip>
  </ctgu:item>
  <ctgu:item>
    <ctgu:name>Jack</ctgu:name>
    <ctgu:ID>2002105102</ctgu:ID>
    <ctgu:Phone>6390002</ctgu:Phone>
    <ctgu:Zip>443002</ctgu:Zip>
  </ctgu:item>
  <ctgu:item>
    <ctgu:name>Kate</ctgu:name>
    <ctgu:ID>2002105103</ctgu:ID>
    <ctgu:Phone>6390003</ctgu:Phone>
    <ctgu:Zip>443002</ctgu:Zip>
  </ctgu:item>
</ctgu:menmber>
```

19.3 使用 Python 处理 XML

在 Python 中提供了许多标准模块用于处理 XML 文档。例如使用 Expat 分析器的 xml.parsers.expat 模块，使用 SAX 分析器的 xml.sax 模块，使用 DOM 的 xml.dom 模块。其中，xml.parsers.expat 和 xml.sax 模块与 HTMLParser 类似，都是基于事件的方式对 XML 文档进行分析。

19.3.1 使用 xml.parsers.expat 处理 XML

在 Python 中使用 xml.parsers.expat 处理 XML 时，应首先使用其 ParserCreate 创建一个 XMLParser 实例对象。其原型如下所示。

```
ParserCreate(encoding, namespace_separator)
```


其参数含义如下。

- encoding: XML 文档的编码, 可选参数。
- namespace_separator: XML 文档的命名空间, 可选参数。

当创建 XMLParser 对象后需要使用 Parse 或者 ParseFile 方法向其传递要处理的 XML 数据或 XML 文档。其原型分别如下所示。

```
Parse(data, isfinal)
ParseFile(file)
```

对于 Parse, 其参数含义如下。

- data: 要进行处理 XML 数据。
- isfinal: 当最后一次调用该方法时, isfinal 应为 True, 可选参数。

对于 ParseFile, 其参数含义如下所示。

- file: 打开的文件对象。

在使用 XMLParser 处理 XML 的过程中, 当遇到相应的事件时, XMLParser 会调用相应的事件处理方法。在使用 XMLParser 时可以通过继承创建新类, 重载需要的处理方法, 也可以直接使用 XMLParser, 自己编写函数, 将其赋值给 XMLParser 的处理方法。当 XMLParser 遇到 XML 文档声明时, 会调用 XmlDeclHandler 方法, 其原型如下所示。

```
XmlDeclHandler(version, encoding, standalone)
```

其参数如下。

- version: XML 规范的版本。
- encoding: XML 文档的编码。
- standalone: XML 文档的 standalone 属性值。

当 XMLParser 遇到文档类型定义开始时, 将调用 StartDoctypeDeclHandler 方法, 当结束时将调用 EndDoctypeDeclHandler 方法。其原型分别如下所示。

```
StartDoctypeDeclHandler(doctypeName, systemId, publicId, has_internal_subset)
EndDoctypeDeclHandler()
```

对于 StartDoctypeDeclHandler, 其参数含义如下。

- doctypeName: DTD 名称。
- systemId: 系统标识。
- publicId: 公共标识。
- has_internal_subset: 如果 XML 文档包含 DTD, 则 has_internal_subset 为真。

当 XMLParser 遇到 DTD 中元素声明时, 将调用 ElementDeclHandler 方法。其原型如下所示。

```
ElementDeclHandler(name, model)
```

其参数含义如下。

- name: 元素名。
- model: 元素内容模型。

当 XMLParser 遇到 DTD 中元素属性声明时, 将调用 AttlistDeclHandler 方法。其原型如下所示。

```
AttlistDeclHandler(ename, attname, type, default, required)
```

其参数含义如下。

- ename: 元素名。
- attname: 属性名。
- type: 元素数据类型。
- default: 属性默认值。
- required: 如果属性为必需的, 则 required 为真。

当 XMLParser 遇到元素开始标记时, 将调用 StartElementHandler 方法, 当遇到结束标记时, 将调用 EndElementHandler 方法。其原型分别如下所示。

```
StartElementHandler(name, attributes)
```

```
EndElementHandler(name)
```

其参数含义如下。

- name: 元素名。
- attributes: 元素属性。

当 XMLParser 遇到 XML 处理指令时, 将调用 ProcessingInstructionHandler 方法。其原型如下所示。

```
ProcessingInstructionHandler(target, data)
```

其参数含义如下。

- target: 指令名称。
- data: 指令数据。

当 XMLParser 遇到字符数据时, 将调用 CharacterDataHandler 方法。其原型如下所示。

```
CharacterDataHandler(data)
```

其参数含义如下。

- data: 字符数据。

当 XMLParser 遇到 XML 文档中的注释时, 将调用 CommentHandler 方法。其原型如下所示。

```
CommentHandler(data)
```

其参数含义如下。

- data: 注释内容。

当 XMLParser 遇到 CDATA 开始时, 将调用 StartCdataSectionHandler 方法, 当遇到 CDATA

结束时将调用 `EndCdataSectionHandler` 方法。

19.3.2 使用 `xml.sax` 处理 XML

与 `xml.parsers.expat` 模块不同, `xml.sax` 模块将分析器和处理器分离了。使用 `xml.sax` 模块时, 可以使用 `make_parser` 函数创建分析器, 它返回一个 `XMLReader` 对象。然后使用 `XMLReader` 对象的 `setContentHandler` 设置 `XMLReader` 对象的 `ContentHandler`。在脚本中通过继承 `ContentHandler` 类, 重载相应的处理方法, 即可对 XML 文档进行处理。如果需要对 DTD 进行处理, 可以使用 `XMLReader` 对象的 `setDTDHandler` 方法。使用 `setDTDHandler` 方法设置 `XMLReader` 对象的 `DTDHandler` 句柄。

`xml.sax` 的分析器在分析 XML 文档时, 在 XML 文档开始时将调用 `ContentHandler` 的 `startDocument` 方法, 在 XML 文档结束时将调用 `ContentHandler` 的 `endDocument` 方法。其原型分别如下所示。

```
startElement(name, attrs)
endElement(name)
```

其参数含义如下。

- `name`: 元素名。
- `attrs`: 元素属性。

如果在 XML 文档中使用了命名空间, 将调用 `startElementNS` 方法和 `endElementNS` 方法。其原型分别如下所示。

```
startElementNS(name, qname, attrs)
endElementNS(name, qname)
```

其参数含义如下。

- `name`: 由 URI 和本地名组成的元组。
- `qname`: 元素名。
- `attrs`: 元素属性。

当 `xml.sax` 的分析器遇到字符数据时, 将调用 `ContentHandler` 的 `characters` 方法。其原型如下所示。

```
characters(content)
```

其参数含义如下。

- `content`: 字符数据内容。

当 `xml.sax` 的分析器遇到 XML 处理指令时, 将调用 `ContentHandler` 的 `processingInstruction` 方法。其原型如下所示。

```
processingInstruction(target, data)
```

其参数含义如下。

- `target`: 指令名称。

- data: 指令数据。

当 xml.sax 的分析器处理 DTD 时, 将调用 DTDHandler 中的方法。在 DTDHandler 中主要提供了用于处理声明的 notationDecl 方法和用于处理分析实体声明 unparsedEntityDecl 方法。其原型分别如下所示。

```
notationDecl(name, publicId, systemId)
unparsedEntityDecl(name, publicId, systemId, ndata)
```

19.3.3 使用 xml.dom 处理 XML

DOM (Document Object Model), 即文档对象模型, 使用树式结构创建 XML 文档。Python 的 xml.dom 模块提供了 DOM 接口, 可以将 XML 文档转为树结构。Python 提供了基本的 DOM 分析系统 minidom 模块和复杂的 DOM 分析系统 pulldom 模块。minidom 模块适用于较小的 XML 文档, 因为它将整个文档读到内存中。而 pulldom 模块则适用于较大的 XML 文档, 因为它不将整个 XML 文档读入内存。

Python 的 xml.dom 中的接口较多, 其常用的有 Element 对象中的如下几种方法。

- getElementsByTagName(): 用于获取标记的分支。
- getElementsByTagNameNS(): 用于获取标记的分支 (使用了命名空间的情况)。
- getAttribute(): 用于获取属性值。
- getAttributeNode(): 用于获取属性节点。
- getAttributeNS(): 用于获取属性值 (使用了命名空间的情况)。
- getAttributeNodeNS(): 用于获取属性节点 (使用了命名空间的情况)。
- removeAttribute(): 用于移除属性。
- removeAttributeNode(): 用于移除属性节点。
- removeAttributeNS(): 用于移除属性 (使用了命名空间的情况)。
- setAttribute(): 用于设置属性。
- setAttributeNS(): 用于设置属性 (使用了命名空间的情况)。
- setAttributeNode(): 用于设置属性节点。
- setAttributeNodeNS(): 用于设置属性节点 (使用了命名空间的情况)。

使用 minidom 时应首先使用 parse, 或 parseString 方法获取 XML 数据。当获取 XML 数据后就可以使用 xml.dom 中的方法对 XML 文档进行处理。

19.4 简单的 RSS 阅读器

RSS (Really Simple Syndication) 是一种描述和同步网站内容的格式。RSS 是基于 XML 的。使用 Python 可以做一个简单的 RSS 阅读器。RSS 有其自己的标准, 可以通过阅读其标

准编写一个脚本来处理网站的 RSS。但鉴于 XML 的良好可读性和良好的结构，只要找到一个简单的 RSS 文件，完全可以写出一个简单的 RSS 阅读器。以 Python 官方网站提供的 RSS 为例，将其打开后，部分内容如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>

<rss version="2.0">
  <channel>
    <title>Python News</title>
    <link>http://www.python.org/</link>
    <description>Python-related news and announcements.
      Python is an interpreted, interactive, object-oriented
      programming language. </description>

    <image>
      <title>Python logo</title>
      <url>http://www.python.org/images/python-logo.gif</url>
      <link>http://www.python.org/</link>
    </image>

    <item>
      <guid>http://www.python.org/news/index.html#Tue24Apr20070447-0400</guid>
      <title>Google Adds Python Support to Google Calendar Developers Guide</title>
      <description><![CDATA[<!--utf-8--><!--0.5-->

        <p>Google, who already support Python in their web apps with their <a
        class="reference" href="http://code.google.com/p/gdata-python-client/">Python Client
        Library</a>, has <a class="reference" href="http://googledataapis.blogspot.com/2007/
        04/python-developers-guide-for-google.html">added coverage of Python</a> to the
        examples and explanations in the <a class="reference" href="http://code.google.com/
        apis/calendar/developers_guide_python.html">Google Calendar Developers Guide</a>.
        Instructional materials for their other REST-based APIs will be forthcoming.</p>]]></
        description>
        <pubDate>Tue, 24 Apr 2007 04:47 -0400</pubDate>
      </item>
      <item>
        <guid>http://www.python.org/news/index.html#Tue24Apr20070347-0400</guid>
        <title>Nigerian 11-Year-Old Students Power Up Their Python-Based
        Laptops</title>
        <description><![CDATA[<!--utf-8--><!--0.5-->

          <p><a class="reference" href="http://news.com.com/2300-1041_3-6175025-4.html?>
```

```

tag=ne.gall.pg">Nigerian youngsters got their hands</a> on the first of millions of 'XO'
laptops, heavily based on Python under Linux. Part of the <a class="reference"
href="http://www.laptop.org">One Laptop per Child</a> project, you can watch <a
class="reference" href="http://www.ivr-usability.com/olpc/olpc.html">a screencast of
the GUI</a> and <a class="reference" href="http://wiki.laptop.org/go/OLPC_Python_
Environment">get involved in the Python side</a> of the project.</p>]]</description>
  <pubDate>Tue, 24 Apr 2007 03:47 -0400</pubDate>
    </item>
    <item>
      <guid>http://www.python.org/news/index.html#Tue24Apr20070147-0400</guid>
      <title>'Computing in Science and Engineering' issue devoted to Python</title>
      <description><![CDATA[<!--utf-8--><!--0.5-->

```

```

      <p>The May/June 2007 issue of <a class="reference" href="http://computer.org/
cise">Computing in Science and Engineering</a>, a joint publication of IEEE and the
American Institute for Physics, is devoted to Python. Some of the articles, covering
everything from IPython, matplotlib and NumPy to robots, the stars and education, are
available for free download.</p>]]></description>

```

```

  <pubDate>Tue, 24 Apr 2007 01:47 -0400</pubDate>

```

```

    </item>

```

```

  省略部分内容

```

```

</channel>

```

```

</rss>

```

可以看到每一条主要的消息处于“item”元素之间。在“item”元素中依次包含“guid”、“title”、“description”、“pubDate”元素。其中“title”元素包含了消息的标题，“description”元素包含了消息的简要描述，“pubDate”元素包含了消息发布的时间。Python官方的RSS文件非常直观，通过查看RSS文件内容即可了解其结构。如下所示的pyRSS.py脚本仅对“title”和“pubDate”元素进行处理，获取其字符数据，即获取消息的标题和发布时间。

```

# -*- coding:utf-8 -*-
# file: pyRSS.py
#
import Tkinter
import urllib
import xml.parsers.expat

class MyXML:
    def __init__(self, edit):
        self.parser = xml.parsers.expat.ParserCreate()
        self.parser.StartElementHandler = self.start
        self.parser.EndElementHandler = self.end
        self.parser.CharacterDataHandler = self.data
        self.title = False
        self.date = False
        self.edit = edit
    def start(self, name, attrs):

```

```

# XML 解析类
# 生成 XMLParser
# 起始标记处理方法
# 结束标记处理方法
# 字符数据处理方法
# 状态标志
# 多行文本框对象
# 起始标记处理方法

```



```

        if name == 'title':
            self.title = True
        elif name == 'pubDate':
            self.date = True
        else:
            pass
    def end(self, name):
        if name == 'title':
            self.title = False
        elif name == 'pubDate':
            self.date = False
        else:
            pass
    def data(self, data):
        if self.title:
            self.edit.insert(Tkinter.END,
                             '*****\n')
            self.edit.insert(Tkinter.END, 'Title: ')
            self.edit.insert(Tkinter.END, data + '\n')
        elif self.date:
            self.edit.insert(Tkinter.END, 'Date: ')
            self.edit.insert(Tkinter.END, data + '\n')
        else:
            pass
    def feed(self, data):
        self.parser.Parse(data, 0)
class Window:
    def __init__(self, root):
        self.root = root
        self.get = Tkinter.Button(root,
                                   text = '获取 RSS', command = self.Get)
        self.get.place(x = 280, y = 15)
        self.frame = Tkinter.Frame(root, bd=2)
        self.scrollbar = Tkinter.Scrollbar(self.frame)
        self.edit = Tkinter.Text(self.frame, yscrollcommand = self.scrollbar.set,
                                   width = 96, height = 32)
        self.scrollbar.config(command=self.edit.yview)
        self.edit.pack(side = Tkinter.LEFT)
        self.scrollbar.pack(side=Tkinter.RIGHT, fill=Tkinter.Y)
        self.frame.place(y = 50)
    def Get(self):
        url = 'http://www.python.org/channews.rdf'
        page = urllib.urlopen(url)
        data = page.read()
        parser = MyXML(self.edit)
        parser.feed(data)
root = Tkinter.Tk()
window = Window(root)
root.minsize(600, 480)

```

判断是否为 title 元素
标志设为真
判断是否为 pubDate
标志设为真
结束标记处理
标志设为假
标志设为假
字符数据处理方法
根据标志状态输出数据
创建组件
打开 URL
读取 URL 内容
生成实例对象
处理 XML 数据

```
root.maxsize(600,480)
root.mainloop()
```

运行 pyRSS.py 脚本后, 单击【获取 RSS】按钮, 如图 19-4 所示。

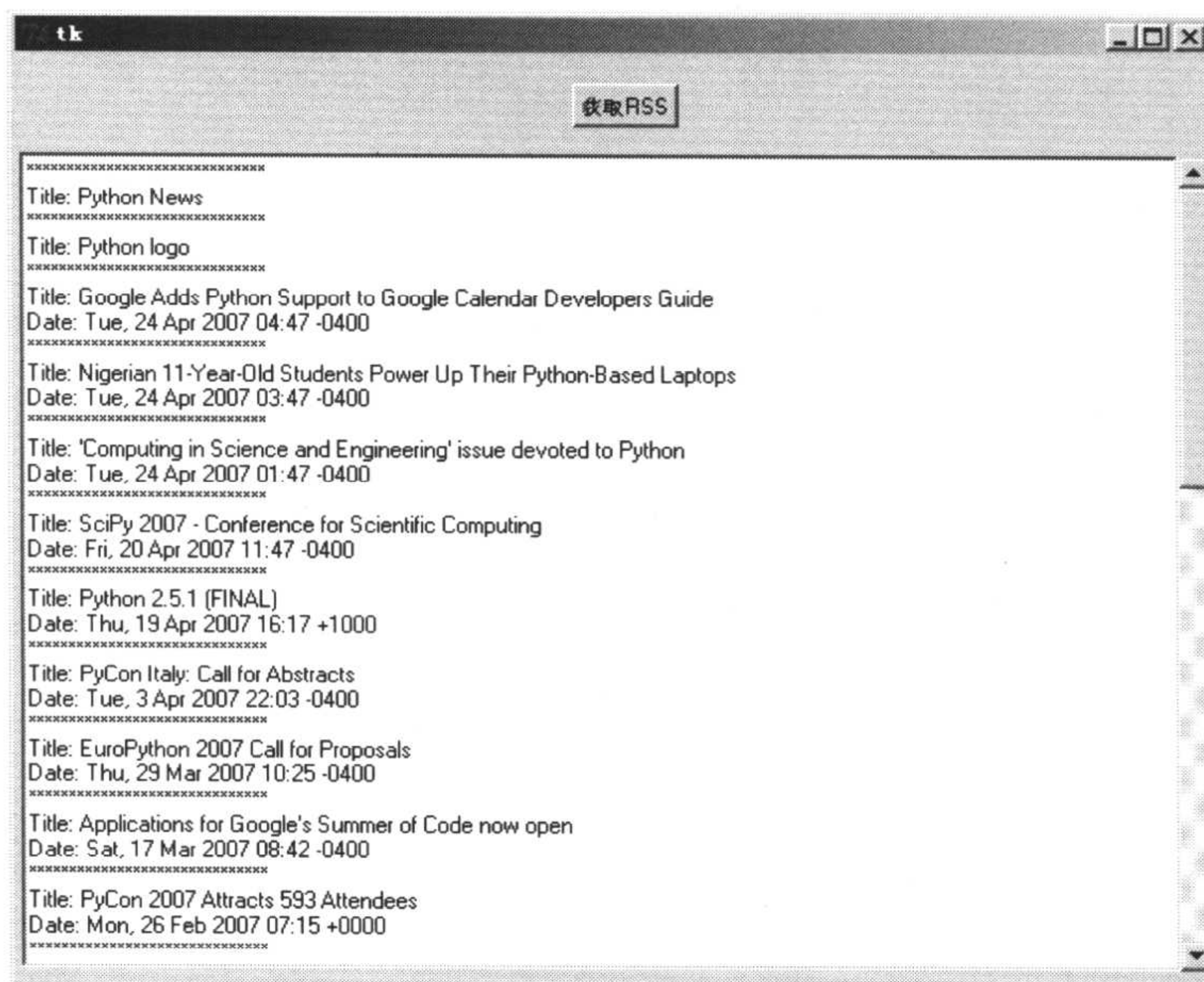
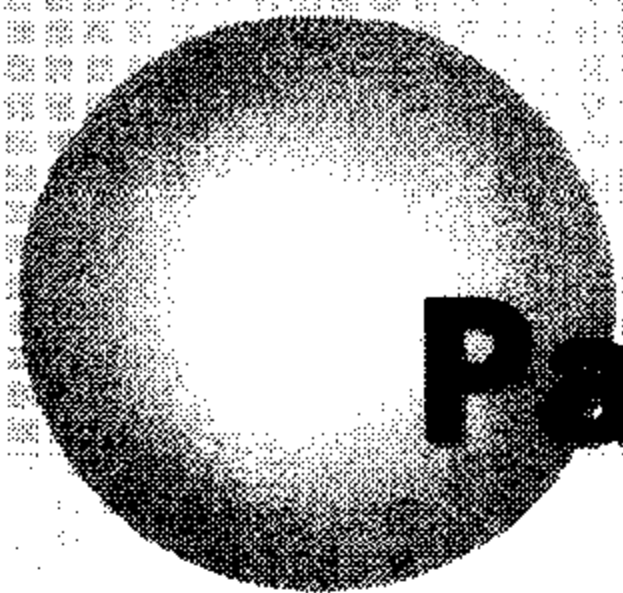


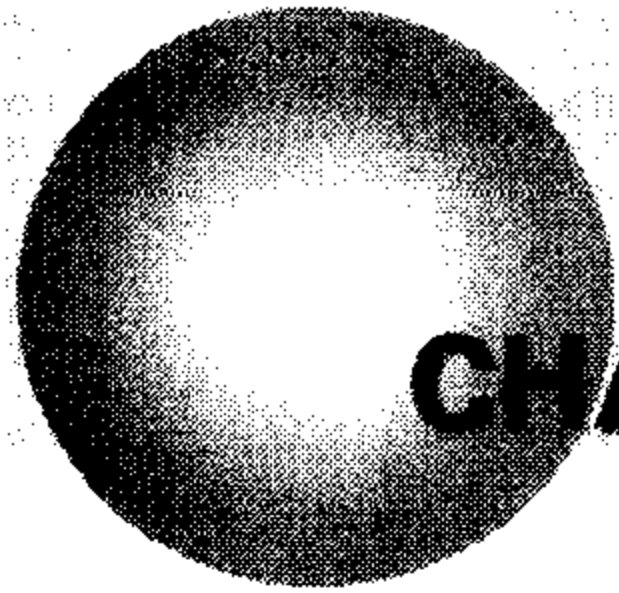
图 19-4 处理 RSS



Part 5

第五篇

多媒体与其他应用



第 20 章 数据结构与算法

数据结构用来描述一种或多种数据元素之间的特定关系。算法是程序设计中
对数据操作的描述。数据结构和算法组成了程序。对于简单的任务，只要使用
编程语言提供的基本数据类型就足够了。而对于较复杂的任务，就需要使用
基本的数据类型构造更加复杂的数据结构。

20.1 表、栈和队列

表、栈和队列都是基本的线性数据结构。由于 Python 设计良好的数据结
构，其列表可以当作表来使用。而且列表的某些特性跟链表相似，在 Python
中表的实现非常简单。对于栈和队列，则可以自己在脚本中构建。

20.1.1 表

表是最基本的数据结构，在 Python 中可以使用列表来创建表。而在 C 语
言中，一般使用数组来创建表。由于使用数组创建表，对表中元素进行插入
和删除操作的开销较大。当插入一个元素时，要先将该元素后的所有元素，
从最后一个元素开始，依次向后移动一个位置。完成元素移动后，再将元
素插入数组中。同样，要删除表中的元素时，首先删除元素，然后将位于
该元素之后的元素从前向后，依次向前移动一个位置。

如果一个表含有元素较多，而要进行插入或删除的位置又比较靠近表的前
端，则移动元素的操作将耗费大量的时间。为了减少插入和删除元素的线性
开销，可以使用链表代替表。在 C 语言中，链表中不仅保存数据，还保存
了指向下一个元素的指针，如图 20-1 所示。当进行插入操作时，要先将
位于插入元素前的元素的指针赋值给插入元素。完成赋值后再将插入元
素的地址赋值给位于其前的元素，如图 20-2 所示。当进行删除元素时，
只需将要删除元素的指针赋值给其前边的元素即可，如图 20-3 所示。

使用链表，可以降低插入、删除元素的线性开销。由于链表中不仅存储了
数据，而且还保存了指向下一个元素的指针，因而使用链表将占用更大的存
储空间。而在 Python 中，列表本身就提供了插入和删除操作。因此，在
Python 中列表也可以充当链表使用，而不用自己构建。

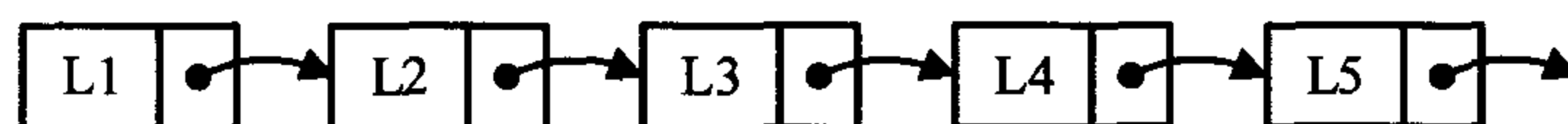


图 20-1 链表

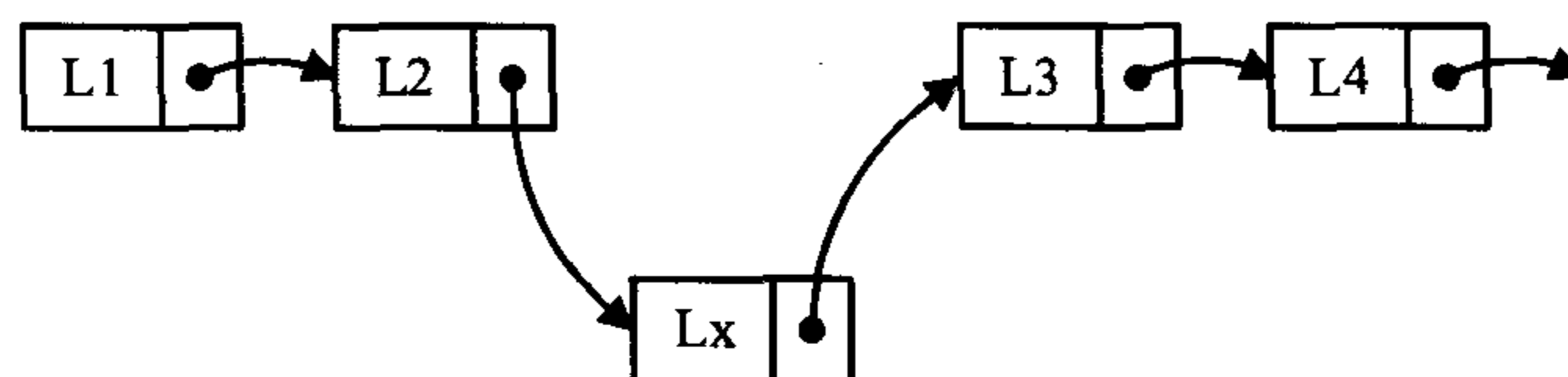


图 20-2 链表的插入操作



图 20-3 链表的删除操作

还有一种链表，称之为双向链表，如图 20-4 所示。双向链表中不仅保存了指向下一元素地址的指针，而且还保存了指向其上一个元素地址的指针。相对于单向链表，双向链表需要更大的存储空间，但使用双向列表可以完成倒序扫描链表。



图 20-4 双向链表

20.1.2 栈

栈可以看作插入和删除在同一位置上进行的表，一般是栈顶。栈的基本操作是进栈和出栈，栈可以看作是一个容器，如图 20-5 所示，先入栈的在容器底部，后入栈的在容器顶部。在出栈的时候，后入栈的先出，而先入栈的后出，因此栈有一个特性叫做后进先出（LIFO）。

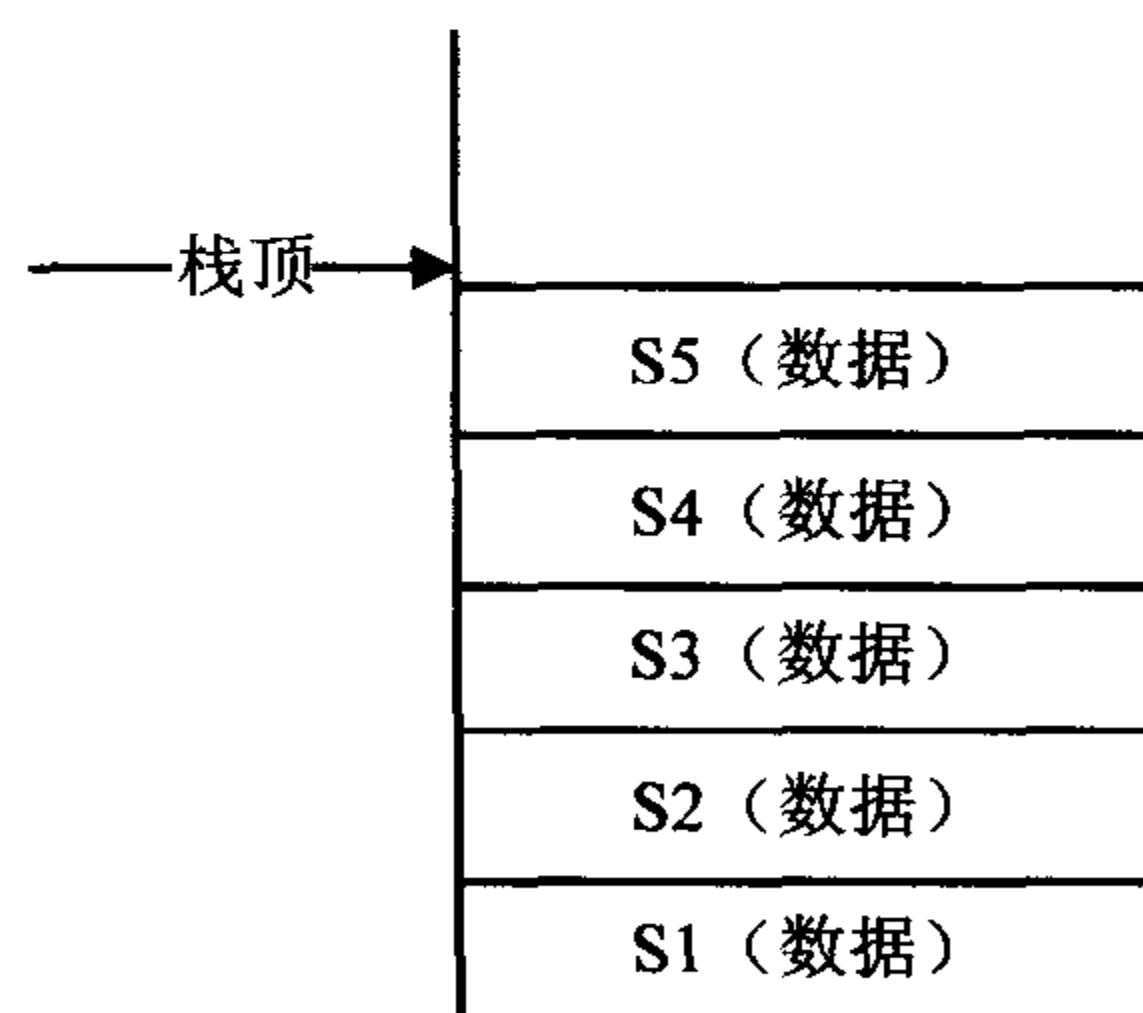


图 20-5 栈

在 Python 中，仍然可以使用列表来存储堆栈数据。通过创建堆栈类，来实现对堆栈进行操作的方法。例如，进栈 PUSH 方法、出栈 POP 方法，编写检查栈是否为满栈，或者是否为

空栈的方法等。如下所示的 pystack.py 在 Python 中创建了一个简单的堆栈结构。

```
# -*- coding:utf-8 -*-
# file: pystack.py
#
class PyStack:                                # 堆栈类
    def __init__(self, size = 20):
        self.stack = []                       # 堆栈列表
        self.size = size                     # 堆栈大小
        self.top = -1                        # 栈顶位置
    def setSize(self, size):                  # 设置堆栈大小
        self.size = size
    def push(self, element):                  # 元素进栈
        if self.isFull():
            raise 'PyStackOverflow'          # 如果栈满, 则引发异常
        else:
            self.stack.append(element)
            self.top = self.top + 1
    def pop(self):                            # 元素出栈
        if self.isEmpty():
            raise 'PyStackUnderflow'         # 如果栈为空, 则引发异常
        else:
            element = self.stack[-1]
            self.top = self.top - 1
            del self.stack[-1]
            return element
    def Top(self):                            # 获取栈顶位置
        return self.top
    def empty(self):                          # 清空栈
        self.stack = []
        self.top = -1
    def isEmpty(self):                        # 是否为空栈
        if self.top == -1:
            return True
        else:
            return False
    def isFull(self):                         # 是否为满栈
        if self.top == self.size - 1:
            return True
        else:
            return False
if __name__ == '__main__':
    stack = PyStack()                         # 创建栈
    for i in range(10):
        stack.push(i)                        # 元素进栈
    print stack.Top()                        # 输出栈顶位置
    for i in range(10):
        print stack.pop()                   # 元素出栈
    stack.empty()                            # 清空栈
    for i in range(21):
```



```
stack.push(i)
```

此处将引发异常

运行 pystack.py 脚本后输出如下所示。

```
9
9
8
7
6
5
4
3
2
1
0
E:\book\code\ystack.py:16: DeprecationWarning: raising a string exception is deprecated
  raise 'PyStackOverflow'
Traceback (most recent call last):
  File "E:\book\code\ystack.py", line 36, in <module>
    stack.push(i)
  File "E:\book\code\ystack.py", line 16, in push
    raise 'PyStackOverflow'
PyStackOverflow
```

20.1.3 队列

队列与栈类似，如图 20-6 所示，但不同的是，队列的出队操作是队首元素进行的删除操作，因而对于队列而言，先入队的元素将先出队。因此队的特性可以称为先进先出（FIFO）。

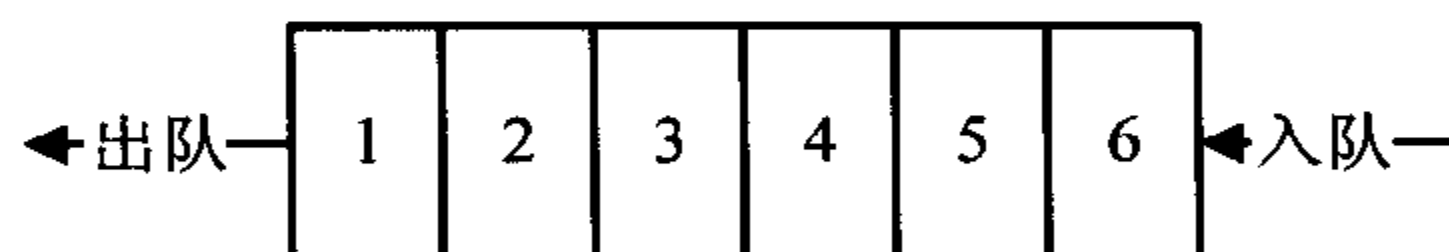


图 20-6 队

和堆栈类似，在 Python 中同样可以使用列表来构建一个队列，并完成对队列的操作。如下所示的 pyqueue.py 脚本创建一个简单的队列。

```
# -*- coding:utf-8 -*-
# file: pyqueue.py
#
class PyQueue:                                     # 创建队列
    def __init__(self, size = 20):
        self.queue = []                           # 队列
        self.size = size                          # 队列大小
        self.end = -1                             # 队列尾
    def setSize(self, size):                       # 设置队列大小
        self.size = size
    def In(self, element):                         # 入队
        if self.end < self.size - 1:
```

第20章 数据结构与算法

```

        self.queue.append(element)
        self.end = self.end + 1
    else:
        raise 'PyQueueFull'
def Out(self):
    if self.end != -1:
        element = self.queue[0]
        self.queue = self.queue[1:]
        self.end = self.end - 1
        return element
    else:
        raise 'PyQueueEmpty'
def End(self):
    return self.end
def empty(self):
    self.queue = []
    self.end = -1
if __name__ == '__main__':
    queue = PyQueue()
    for i in range(10):
        queue.In(i)
    print queue.End()
    for i in range(10):
        print queue.Out()
    for i in range(20):
        queue.In(i)
    queue.empty()
    for i in range(20):
        print queue.Out()

```

如果队列满, 则引发异常
出队
如果队列为空, 则引发异常
输出队尾
清除队列
元素入队
元素出队
元素入队
清空队列
此处将引发异常

运行 pyqueue.py 脚本后输出如下所示。

```

9
0
1
2
3
4
5
6
7
8
9
E:\book\code\pyqueue.py:24: DeprecationWarning: raising a string exceptio
n is deprecated
    raise 'PyQueueEmpty'
Traceback (most recent call last):
  File "E:\book\code\pyqueue.py", line 41, in <module>
    print queue.Out()
  File "E:\book\code\pyqueue.py", line 24, in Out

```

如果队列为空, 则引发异常
此处将引发异常


```
raise 'PyQueueEmpty'
PyQueueEmpty
```

```
# 如果队列为空，则引发异常
```

20.2 树和图

树和前边所讲的数据结构不同，树不是线性的。在处理较多数据时，使用线性结构较慢，而使用树结构则可以提高处理速度。不过树的构建相对于线性的表、堆栈和队列等较为复杂。

20.2.1 树

树是一种非线性的数据结构，如图 20-7 所示。之所以称为树，是因为其形状像一棵倒置的树。每棵树都有一个根节点，如图 20-7 所示的树中，Root 为根节点。A、B、C 为 Root 的儿子，Root 为 A、B、C 的父亲。A、B、C 为兄弟。同样，A 为 D、E 的父亲，D、E 为 A 的儿子，D、E 为兄弟。D、E 为 Root 的孙子，Root 为 D、E 的祖父。在树中，如果一个元素没有儿子，则称为树的叶子。

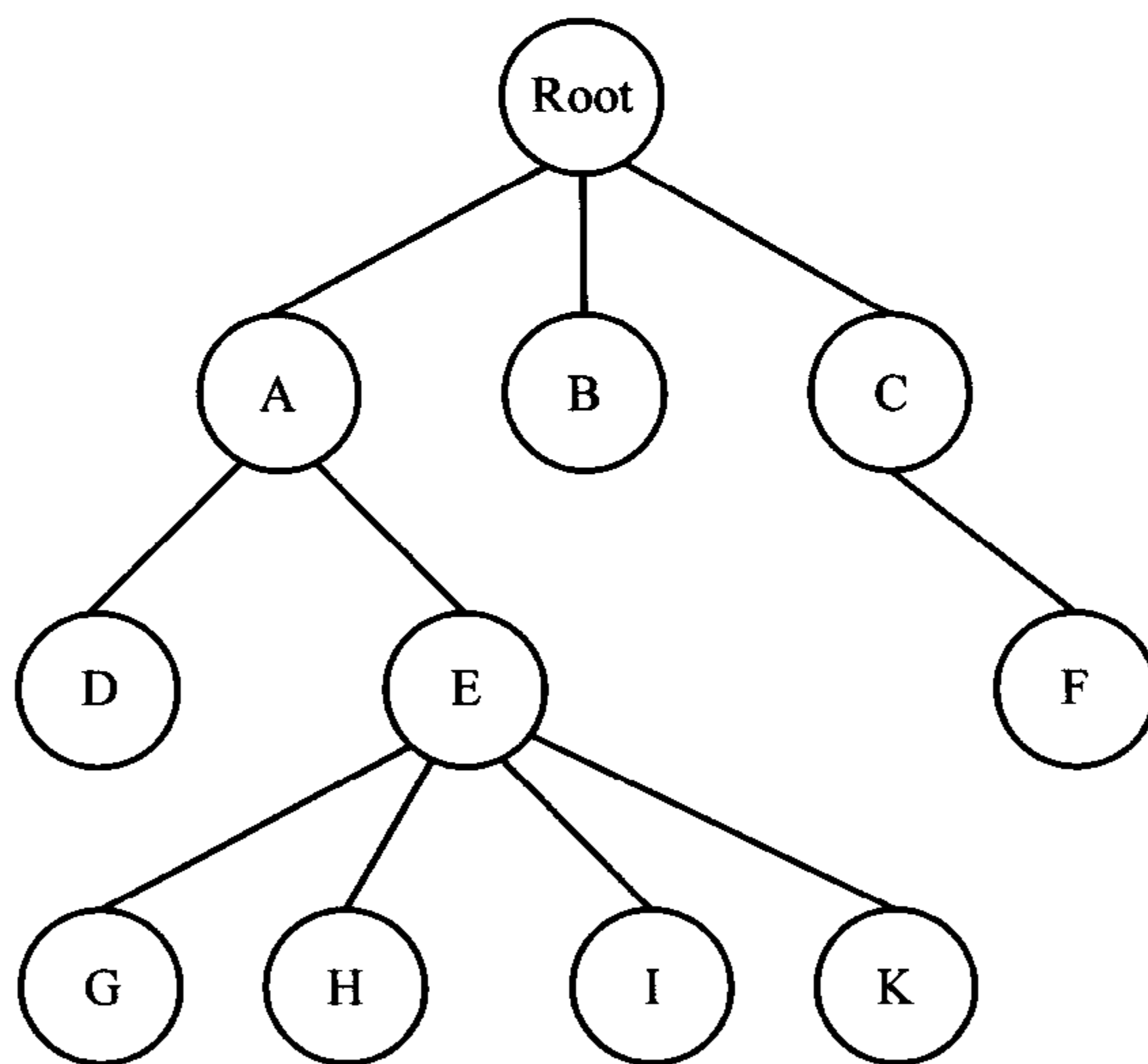


图 20-7 树

在 Python 中树的实现可以使用列表或者类的方式。使用列表的方式较为简便，但树的构建较为复杂。使用类的方式构建树时，需要首先确定树中的节点所能拥有的最大儿子数。因为每个节点所拥有的儿子数并不一定相同，因此使用类的方法将占用更大的存储空间。如下所示的 pytree.py 脚本，以列表的形式构建了图 20-7 所示的树。

```
# -*- coding:utf-8 -*-
# file: pytree.py
#
G = [ 'G', [] ]
```

```
# 构造叶子 G，树中每个元素都由该元素的值和该元素的儿子列表组成
```

```

H = [ 'H', [] ]           # 构造叶子 H
I = [ 'I', [] ]           # 构造叶子 I
K = [ 'K', [] ]           # 构造叶子 K
E = [ 'E', [ G, H, I, K ] ] # 构造 E 节点
D = [ 'D', [] ]           # 构造叶子 D
F = [ 'F', [] ]           # 构造叶子 F
A = [ 'A', [ D, E ] ]     # 构造 A 节点
B = [ 'B', [] ]           # 构造叶子 B
C = [ 'C', [ F ] ]        # 构造 C 节点
Root = [ 'Root', [ A, B, C ] ] # 构造树根
print Root

```

20.2.2 二叉树

二叉树是一类比较特殊的树，在二叉树中每个节点最多只有两个儿子，分为左和右，如图 20-8 所示。相对于树而言，二叉树的构建和使用都要简单得多，而且任何一棵树，都可以通过变换转换成一棵二叉树。

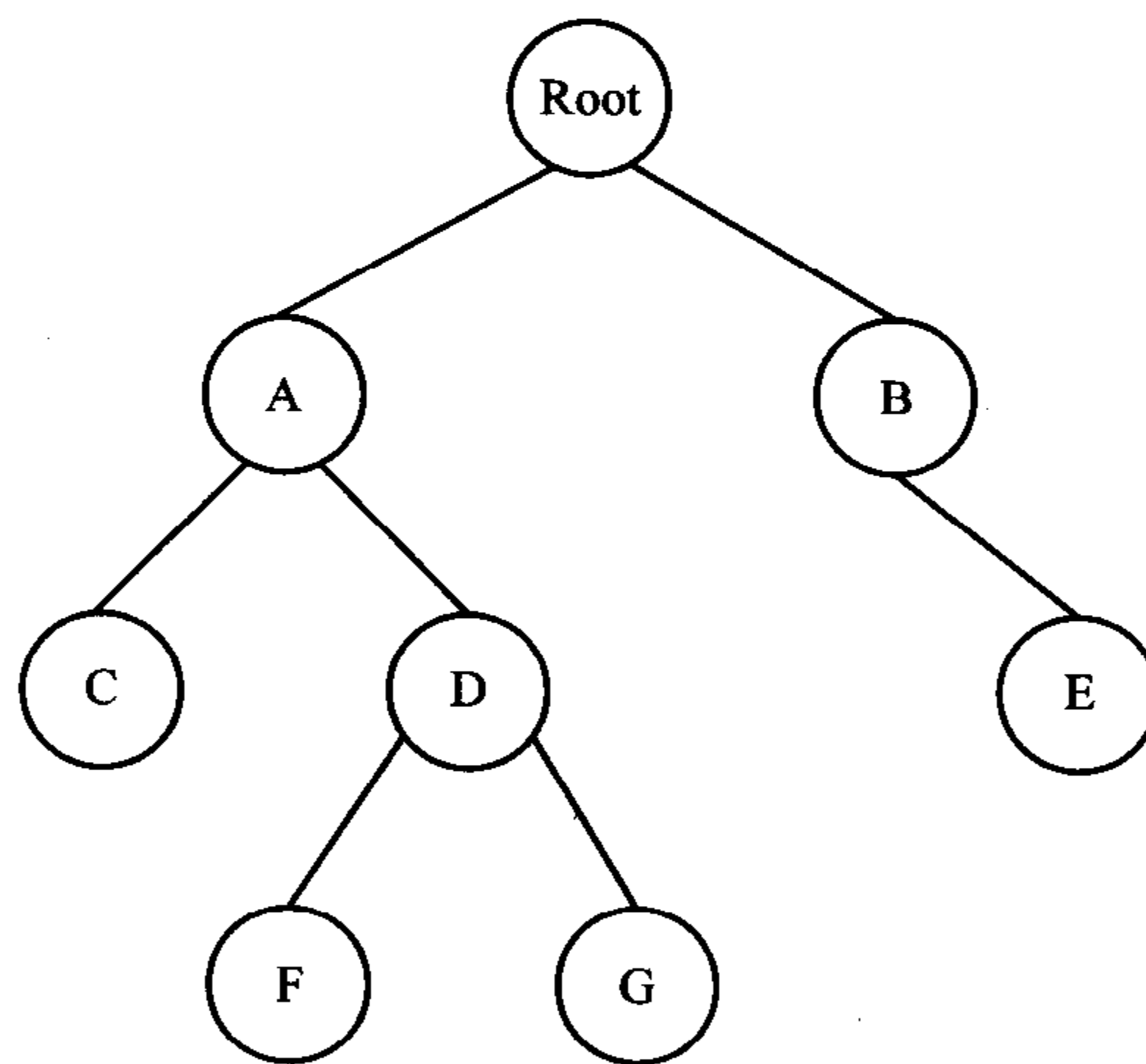


图 20-8 二叉树

在 Python 中，二叉树的构建和树一样，可以使用列表或者类的方式。由于二叉树中的节点具有确定的儿子数，因此，使用类的方式要更为简便。如下所示的 pybtree.py 用比较简单的方式生成了如图 20-8 所示的树。

```

# -*- coding:utf-8 -*-
# file: pybtree.py
#
class BTree:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None

```

二叉树节点
 # 初始化函数
 # 左儿子
 # 节点值
 # 右儿子


```

def insertLeft(self, value):
    self.left = BTree(value)
    return self.left
def insertRight(self, value):
    self.right = BTree(value)
    return self.right
def show(self):
    print self.data
if __name__ == '__main__':
    Root = BTree('Root')
    A = Root.insertLeft('A')
    C = A.insertLeft('C')
    D = A.insertRight('D')
    F = D.insertLeft('F')
    G = D.insertRight('G')
    B = Root.insertRight('B')
    E = B.insertRight('E')
    Root.show()
    Root.left.show()
    Root.right.show()
    A = Root.left
    A.left.show()
    Root.left.right.show()

```

向左子树中插入节点

向右子树中插入节点

输出节点数据

根节点

向根节点中插入 A 节点

向 A 节点中插入 C 节点

向 A 节点中插入 D 节点

向 D 节点中插入 F 节点

向 D 节点中插入 G 节点

向根节点中插入 B 节点

向 B 节点中插入 E 节点

输出节点数据

当创建好一棵二叉树后，可以按照一定的顺序对树中所有的元素进行遍历。按照先左后右，树的遍历方法有 3 种：先序遍历、中序遍历和后序遍历。其中，先序遍历的次序是：如果二叉树不为空，则访问根节点，然后访问左子树，最后访问右子树；否则，程序退出。中序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问根节点，最后访问右子树；否则，程序退出。后序遍历的次序是：如果二叉树不为空，则先访问左子树，然后访问右子树，最后访问根节点。如下所示的 TreeTraversal.py 脚本使用 3 种遍历方式遍历如图 20-8 所示的树。

```

# -*- coding:utf-8 -*-
# file: TreeTraversal.py
#
class BTree:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None
    def insertLeft(self, value):
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):
        self.right = BTree(value)
        return self.right
    def show(self):
        print self.data

```

二叉树节点

初始化函数

左儿子

节点值

右儿子

向左子树中插入节点

向右子树中插入节点

输出节点数据

```

def preorder(node):
    if node.data:
        node.show()
        if node.left:
            preorder(node.left)
        if node.right:
            preorder(node.right)
def inorder(node):
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)
def postorder(node):
    if node.data:
        if node.left:
            postorder(node.left)
        if node.right:
            postorder(node.right)
        node.show()
if __name__ == '__main__':
    Root = BTree('Root')
    A = Root.insertLeft('A')
    C = A.insertLeft('C')
    D = A.insertRight('D')
    F = D.insertLeft('F')
    G = D.insertRight('G')
    B = Root.insertRight('B')
    E = B.insertRight('E')
    print '*****'
    print 'Binary Tree Pre-Traversal'
    print '*****'
    preorder(Root)
    print '*****'
    print 'Binary Tree In-Traversal'
    print '*****'
    inorder(Root)
    print '*****'
    print 'Binary Tree Post-Traversal'
    print '*****'
    postorder(Root)

```

先序遍历

中序遍历

后序遍历

构建树

对树进行先序遍历

对树进行中序遍历

对树进行后序遍历

运行 TreeTraversal.py 脚本后输出如下所示。

```

*****
Binary Tree Pre-Traversal
*****
Root
A
C

```



```
D
F
G
B
E
*****
Binary Tree In-Traversal
*****
C
A
F
D
G
Root
B
E
*****
Binary Tree Post-Traversal
*****
C
F
G
D
A
E
B
Root
```

20.2.3 图

图也是非线性的数据结构，是由顶点和边组成的。如果图中的的顶点是有序的，那么图是有方向的，称之为有向图，如图 20-9 所示；否则，图是无方向的，称之为无向图。在图中，由顶点组成的序列称为路径。

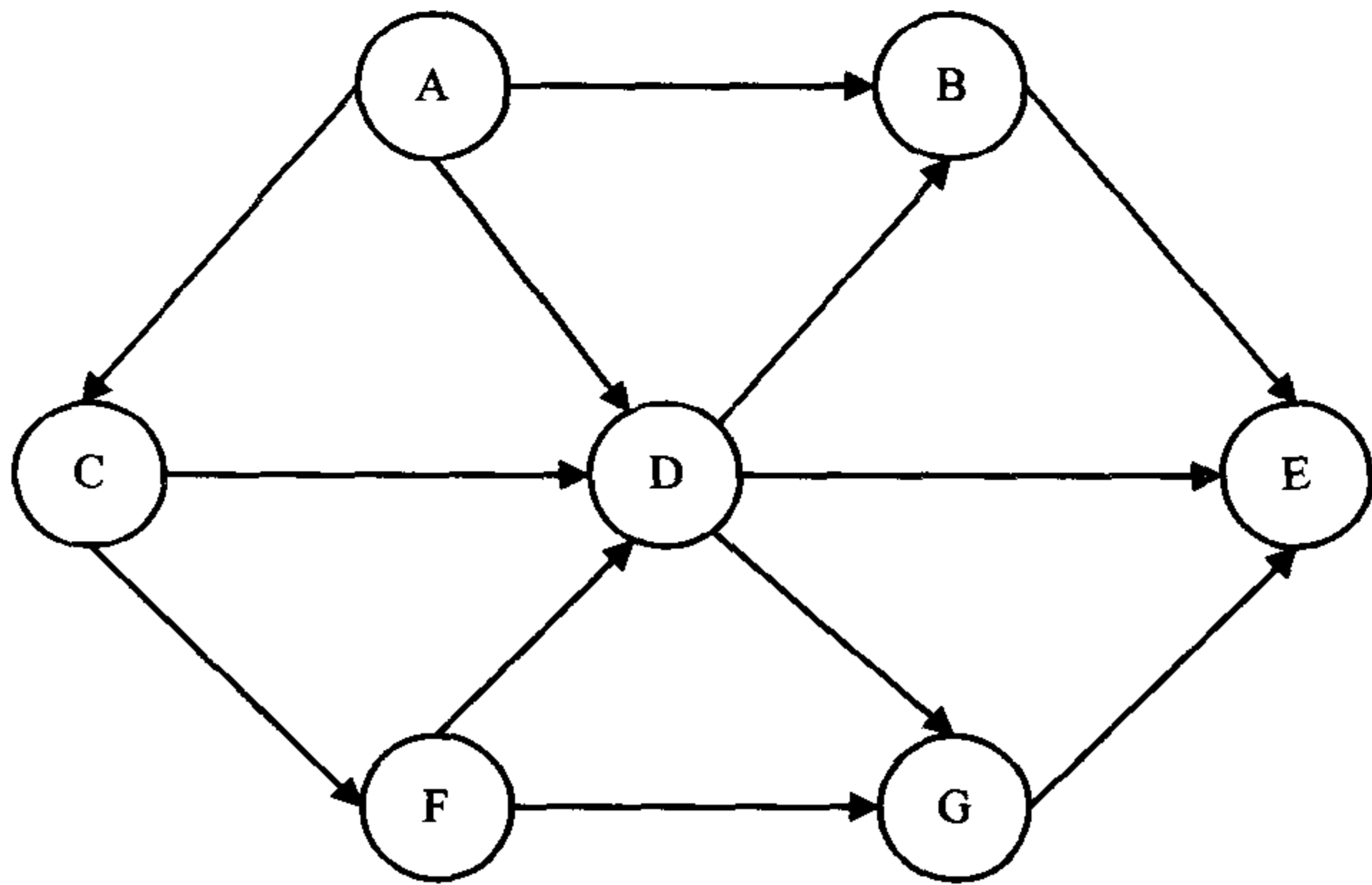


图 20-9 有向图

图和树相比，少了树明显的层次结构。在 Python 中可以采用字典的方式来创建图，图中的每个元素是字典中的键，该元素所指向的图中其他元素则组成键的值。同树一样，对于图来说也可以对其进行遍历。除了遍历以外，可以在图中搜索所有的从一个顶点到另一个顶点的路径。图中的顶点可以看作是城市，路径可以看作是城市到城市之间的公路。因此，通过搜索所有路径，可以找到一个顶点到另一个顶点的最短路径，即城市到城市间的最短路线。如下所示的 pygraph.py 使用字典的方式构建了如图 20-9 所示的有向图，并搜索图中的路径。

```
# -*- coding:utf-8 -*-
# file: pygraph.py
#
def searchGraph(graph, start, end):                                # 搜索树
    results = []                                                  # 路径列表
    generatePath(graph, [start], end, results)                   # 生成路径
    results.sort( lambda x, y:cmp(len(x), len(y)))               # 按路径长短排序
    return results
def generatePath(graph, path, end, results):                      # 生成路径
    state = path[-1]
    if state == end:
        results.append(path)
    else:
        for arc in graph[state]:
            if arc not in path:
                generatePath(graph, path + [arc], end, results)
if __name__ == '__main__':
    Graph = {'A': ['B', 'C', 'D'],                                # 构建树
            'B': ['E'],
            'C': ['D', 'F'],
            'D': ['B', 'E', 'G'],
            'E': [],
            'F': ['D', 'G'],
            'G': ['E']}
    r = searchGraph(Graph, 'A', 'D')                              # 搜索 A 到 D 的所有路径
    print '*****'
    print '    path A to D'
    print '*****'
    for i in r:
        print i
    r = searchGraph(Graph, 'A', 'E')                              # 搜索 A 到 E 的所有路径
    print '*****'
    print '    path A to E'
    print '*****'
    for i in r:
        print i
    r = searchGraph(Graph, 'C', 'E')                              # 搜索 C 到 E 的所有路径
    print '*****'
```



```
print '    path C to E'
print '*****'
for i in r:
    print i
```

运行 pygraph.py 脚本后输出如下所示。

```
*****
    path A to D
*****
['A', 'D']
['A', 'C', 'D']
['A', 'C', 'F', 'D']
*****
    path A to E
*****
['A', 'B', 'E']
['A', 'D', 'E']
['A', 'C', 'D', 'E']
['A', 'D', 'B', 'E']
['A', 'D', 'G', 'E']
['A', 'C', 'D', 'B', 'E']
['A', 'C', 'D', 'G', 'E']
['A', 'C', 'F', 'D', 'E']
['A', 'C', 'F', 'G', 'E']
['A', 'C', 'F', 'D', 'B', 'E']
['A', 'C', 'F', 'D', 'G', 'E']
*****
    path C to E
*****
['C', 'D', 'E']
['C', 'D', 'B', 'E']
['C', 'D', 'G', 'E']
['C', 'F', 'D', 'E']
['C', 'F', 'G', 'E']
['C', 'F', 'D', 'B', 'E']
['C', 'F', 'D', 'G', 'E']
```

20.3 查找与排序

查找和排序是最基本的算法。在很多脚本中都会用到查找和排序，在前边章节的例子中多次使用 Python 的函数查找字符串中的子字符串。尽管 Python 提供了用于查找和排序的函数能够满足绝大多数需求，但还是有必要了解最基本的查找和排序算法。

20.3.1 查找

基本的查找方法有顺序查找、二分查找和分块查找。其中，顺序查找是最简单的查找方

法，就是按数据排列的顺序依次查找，直到找到所查找的数据为止。二分查找是首先对要进行查找的数据进行排序，有按大小顺序排好的 9 个数字，如图 20-10 所示。如果要查找数字 5，首先与中间值 10 进行比较，5 小于 10，于是对序列的前半部分 1~9 进行查找。此时，中间值为 5，恰好为要找的数字，查找结束。

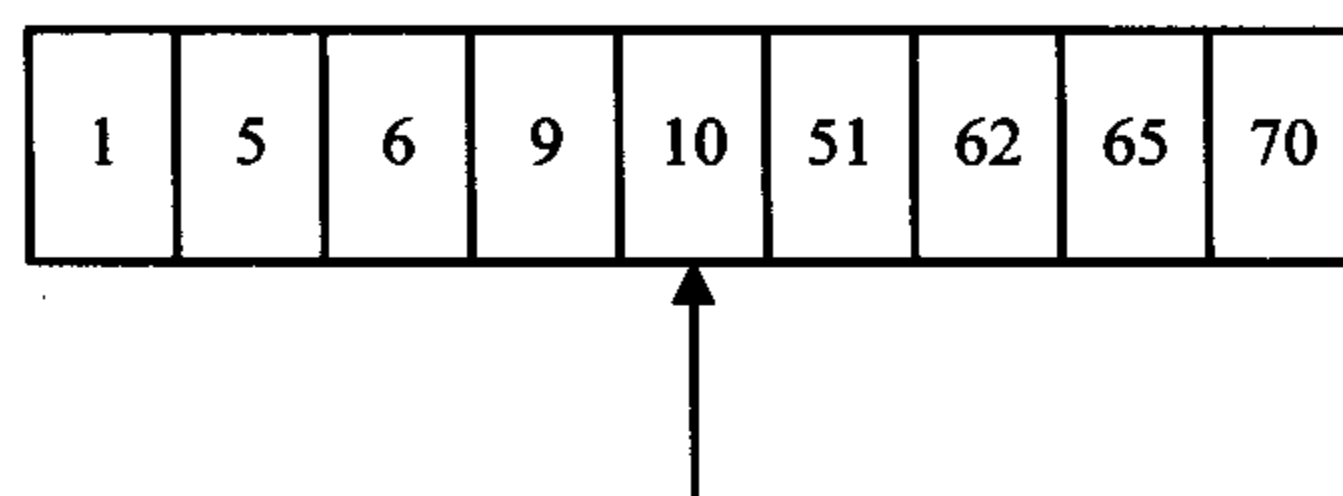


图 20-10 二分查找

分块查找是顺序查找和二分查找之间的一种查找方法。使用分块查找时首先对查找表建立一个索引表，再进行分块查找。建立索引表时，首先对查找表进行分块，要求“分块有序”，即块内关键字不一定有序，但分块之间有大小顺序。索引表是抽取各块中的最大关键字及其起始位置构成的，如图 20-11 所示。分块查找分两步进行，首先查找索引表，因为索引表是有序的，查找索引表时可以使用二分查找。查找完索引表以后，就确定了要查找的数据所在的分块，然后在该分块中进行顺序查找。

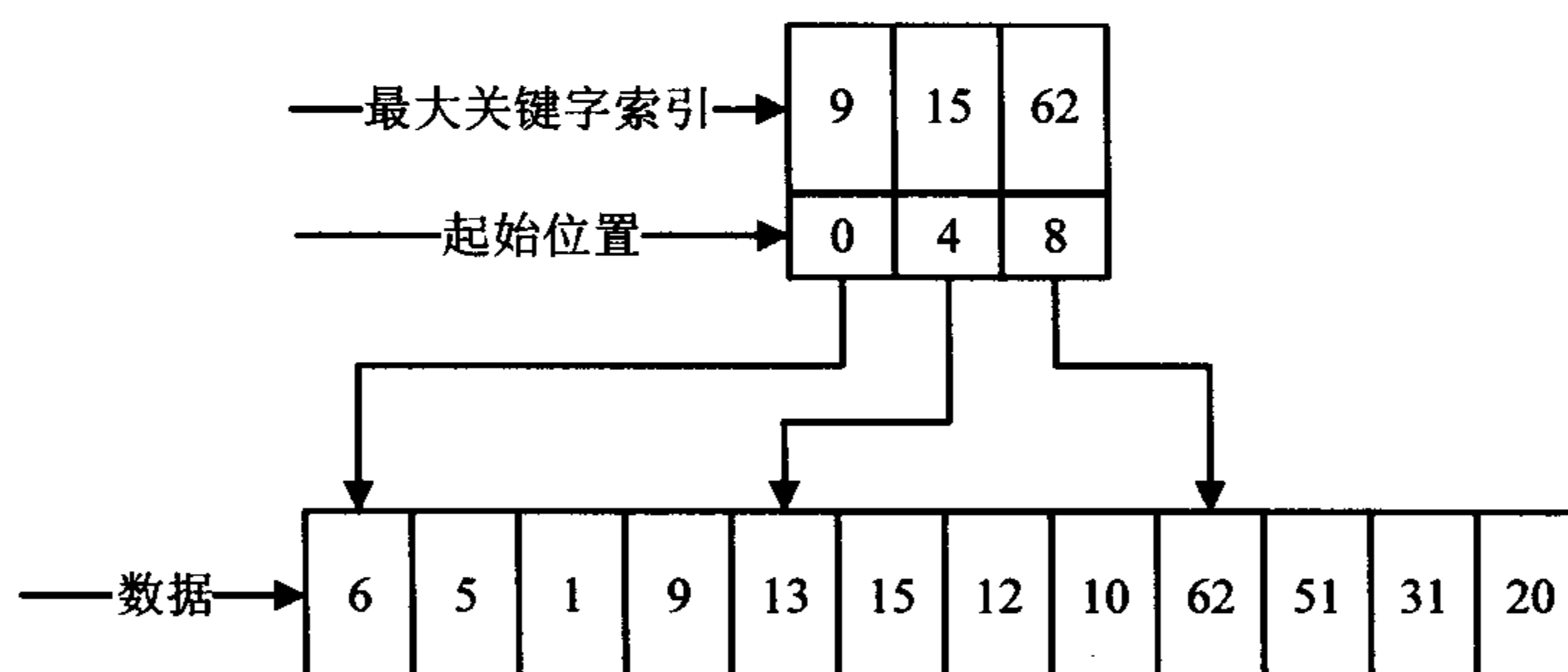


图 20-11 分块查找

如下所示的 pyBinarySearch.py 对一个有序列表使用二分查找。

```
# -*- coding:utf-8 -*-
# file: pyBinarySearch.py
#
def BinarySearch(l, key):                                     # 二分查找
    low = 0
    high = len(l) - 1
    i = 0
    while (low <= high):
        i = i + 1
        mid = (high + low) / 2
        if (l[mid] < key):
```



```

        low = mid + 1;
    elif (l[mid] > key):
        high = mid - 1;
    else:
        print 'use %d time(s)' % i
        return mid
    return -1
if __name__ == '__main__':
    l = [1, 5, 6, 9, 10, 51, 62, 65, 70]
    print BinarySearch(l, 5)
    print BinarySearch(l, 10)
    print BinarySearch(l, 65)
    print BinarySearch(l, 70)

```

构造列表
在列表中查找

运行脚本后输出如下所示。

```

use 2 time(s)
1
use 1 time(s)
4
use 3 time(s)
7
use 4 time(s)
8

```

20.3.2 排序

排序相对于查找来说要复杂得多，排序的方法也较多，有冒泡法排序、希尔排序、二叉树排序和快速排序等。其中二叉树排序是比较有意思的一种排序方法，而且也便于操作。二叉树排序的过程主要是二叉树的建立和遍历的过程。例如有一组数据“3, 5, 7, 20, 43, 2, 15, 30”，则二叉树的建立过程如下所示。

- (1) 首先将第一个数据 3 放入根节点。
- (2) 将数据 5 与根节点中的数据 3 比较，由于 5 大于 3，则将 5 放入 3 的右子树中。
- (3) 将数据 7 与根节点中的数据 3 比较，由于 7 大于 3，则应将 7 放入 3 的右子树中，由于 3 已经有右儿子 5，则将 7 与 5 进行比较，因为 7 大于 5，应将 7 放入 5 的右子树中。
- (4) 将数据 20 与根节点 3 进行比较，由于 20 大于 3，则应将 7 放入 3 的右子树，重复比较，最终将 20 放到 7 的右子树中。
- (5) 将数据 43 与树中的节点值进行比较，最终将其放入 20 的右子树中。
- (6) 将数据 2 与根节点 3 进行比较，由于 2 小于 3，则应将 2 放入 3 的左子树中。
- (7) 同样地对数据 15 和 30 进行处理，最终形成如图 20-12 所示的树。

当树创建好后，对树进行中序遍历后的结果就是对数据从小到大的排序。如果要从大到小进行排序，则可以先从右子树开始进行中序遍历。如下所示的 pySort.py 脚本采用二叉树排序的方式对数据进行排序。

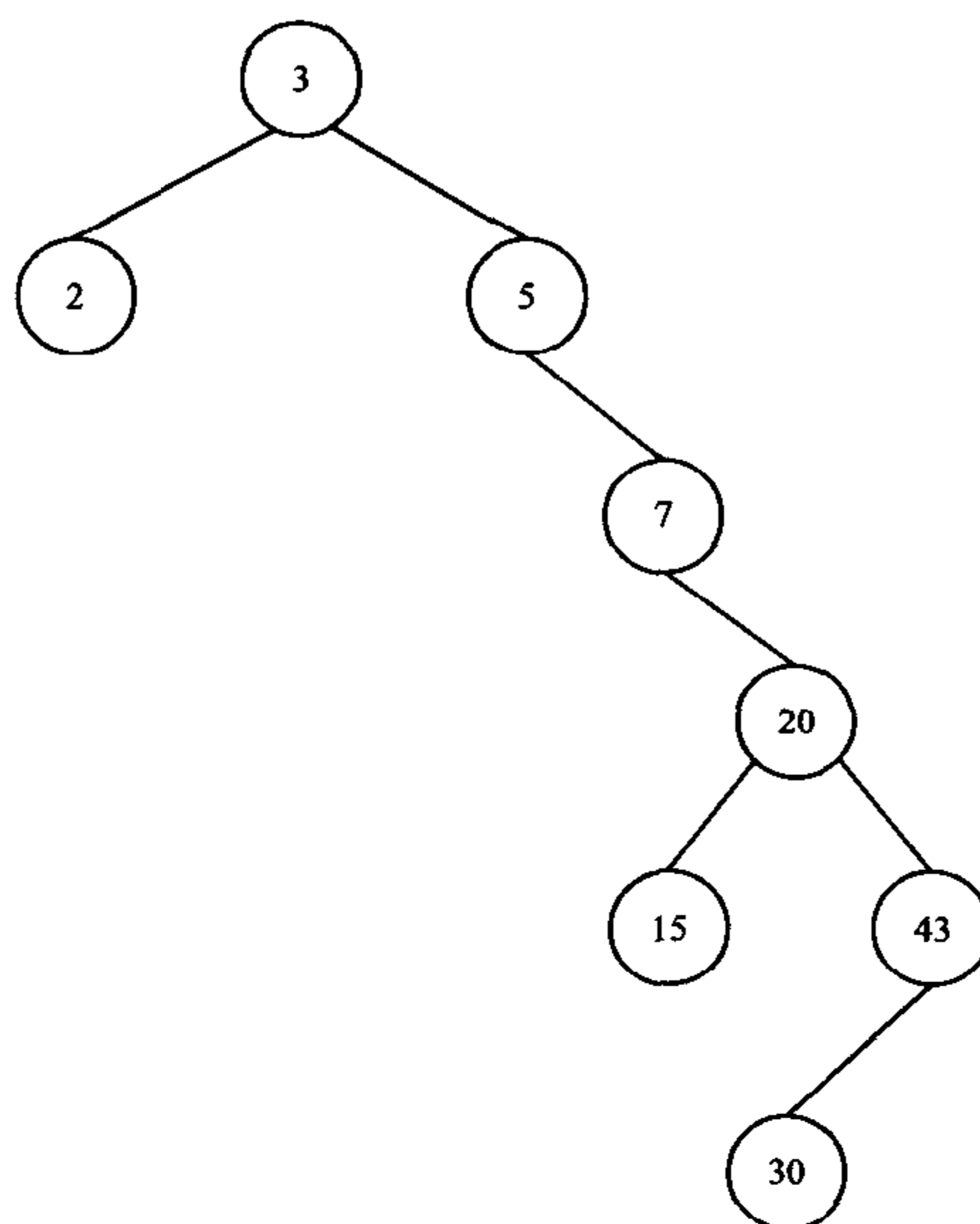


图 20-12 二叉树排序

```

# -*- coding:utf-8 -*-
# file: pySort.py
#
class BTree:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None
    def insertLeft(self, value):
        self.left = BTree(value)
        return self.left
    def insertRight(self, value):
        self.right = BTree(value)
        return self.right
    def show(self):
        print self.data
def inorder(node):
    if node.data:
        if node.left:
            inorder(node.left)
        node.show()
        if node.right:
            inorder(node.right)
def rinorder(node):
    if node.data:
        if node.right:
            rinorder(node.right)
        node.show()
        if node.left:

```

二叉树节点
 # 初始化函数
 # 左儿子
 # 节点值
 # 右儿子
 # 向左子树中插入节点
 # 向右子树中插入节点
 # 输出节点数据
 # 中序遍历
 # 中序遍历


```

        rinorder(node.left)
def insert(node, value):
    if value > node.data:
        if node.right:
            insert(node.right, value)
        else:
            node.insertRight(value)
    else:
        if node.left:
            insert(node.left, value)
        else:
            node.insertLeft(value)
if __name__ == '__main__':
    l = [3, 5, 7, 20, 43, 2, 15, 30]
    Root = BTree(l[0])
    node = Root
    for i in range(1, len(l)):
        insert(Root, l[i])
    print '*****'
    print '        从小到大'
    print '*****'
    inorder(Root)
    print '*****'
    print '        从大到小'
    print '*****'
    rinorder(Root)

```

根节点

运行 pySort.py 脚本后输出如下所示。

```

*****
        从小到大
*****
2
3
5
7
15
20
30
43
*****
        从大到小
*****
43
30
20
15
7
5
3
2

```

第 21 章 科学计算

科学计算是计算机应用的主要内容之一。开源软件 Scilab 和商业软件 Matlab 都是以科学计算为主的应用软件。使用 Python 同样可以进行矩阵运算、数值分析等。

21.1 NumPy 和 SciPy 简介

NumPy 和 SciPy 是 Python 中用以实现科学计算的模块包。NumPy 主要提供了数组对象、基本的数组函数和傅里叶变换的相关函数。SciPy 依赖于 NumPy, SciPy 提供了更多的计算工具, 例如, 绘制图形等。

21.1.1 安装 NumPy 和 SciPy

NumPy 和 SciPy 都提供了 Windows 下的安装包。在 Windows 下安装 NumPy 和 SciPy 较为简单, 只需根据所安装的 Python 版本选择相应的安装文件即可。

1. 安装 NumPy

以 Python 2.5 为例, 在 Windows 下安装 NumPy 的步骤如下所示。

- (1) 从 <http://numpy.scipy.org> 下载 NumPy 在 Windows 下的安装包 numpy-1.0.win32-py2.5.exe。
- (2) 双击运行安装程序, 如图 21-1 所示。
- (3) 单击【下一步】按钮, 将出现如图 21-2 所示的安装路径界面。

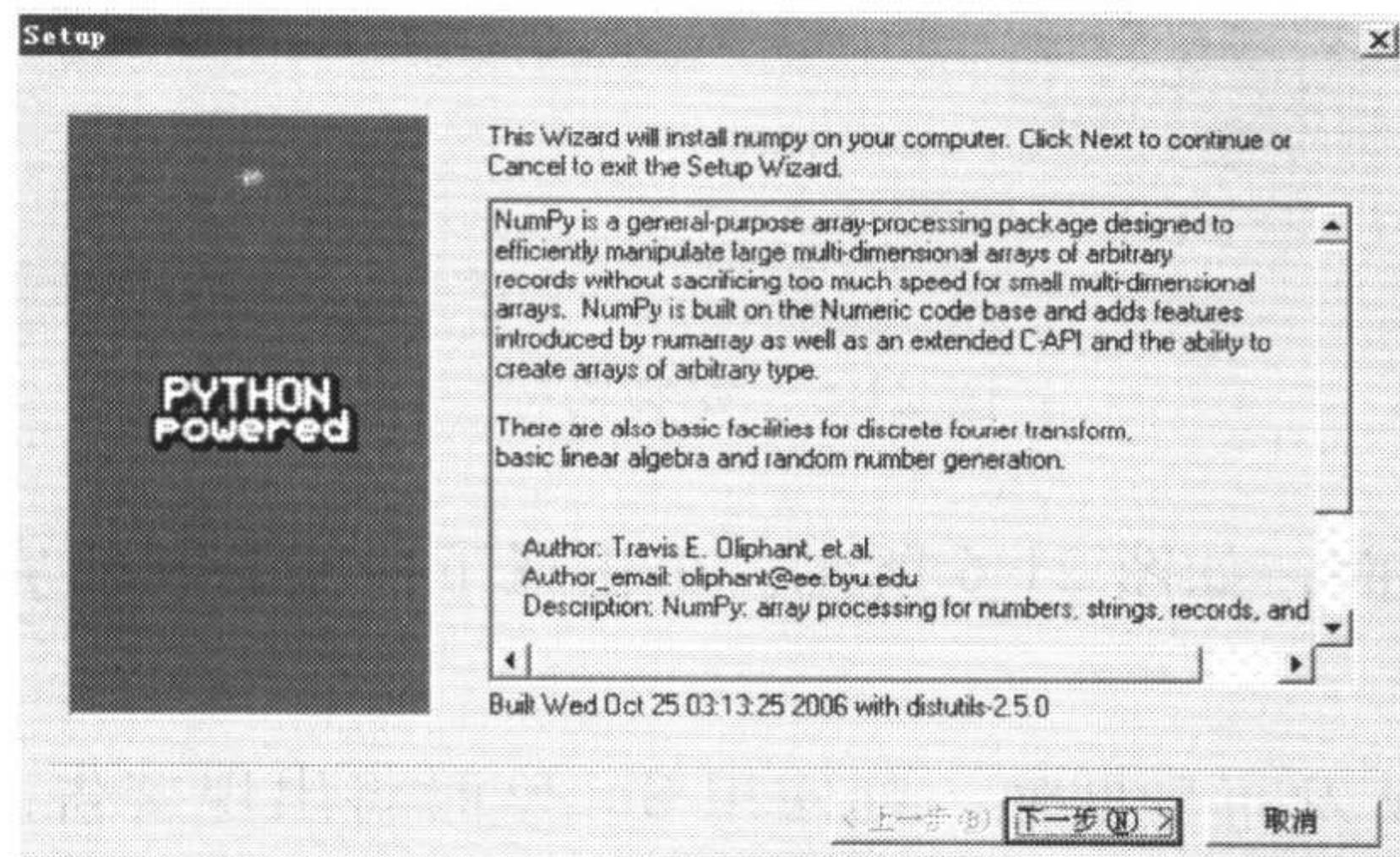


图 21-1 NumPy 安装程序

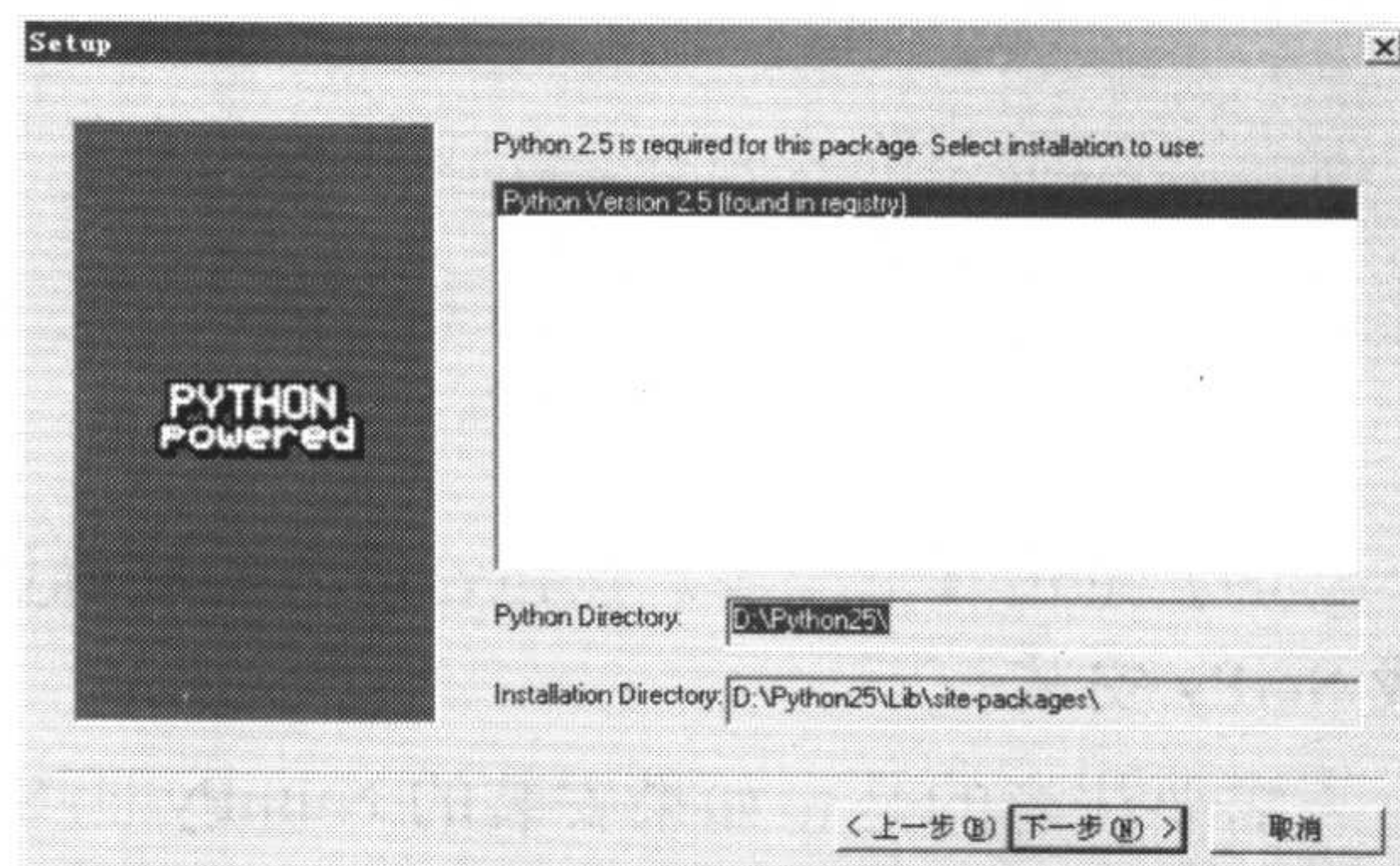


图 21-2 安装路径

(4) 单击【下一步】按钮，进入安装确认界面，如图 21-3 所示。单击【下一步】按钮，完成 NumPy 安装。

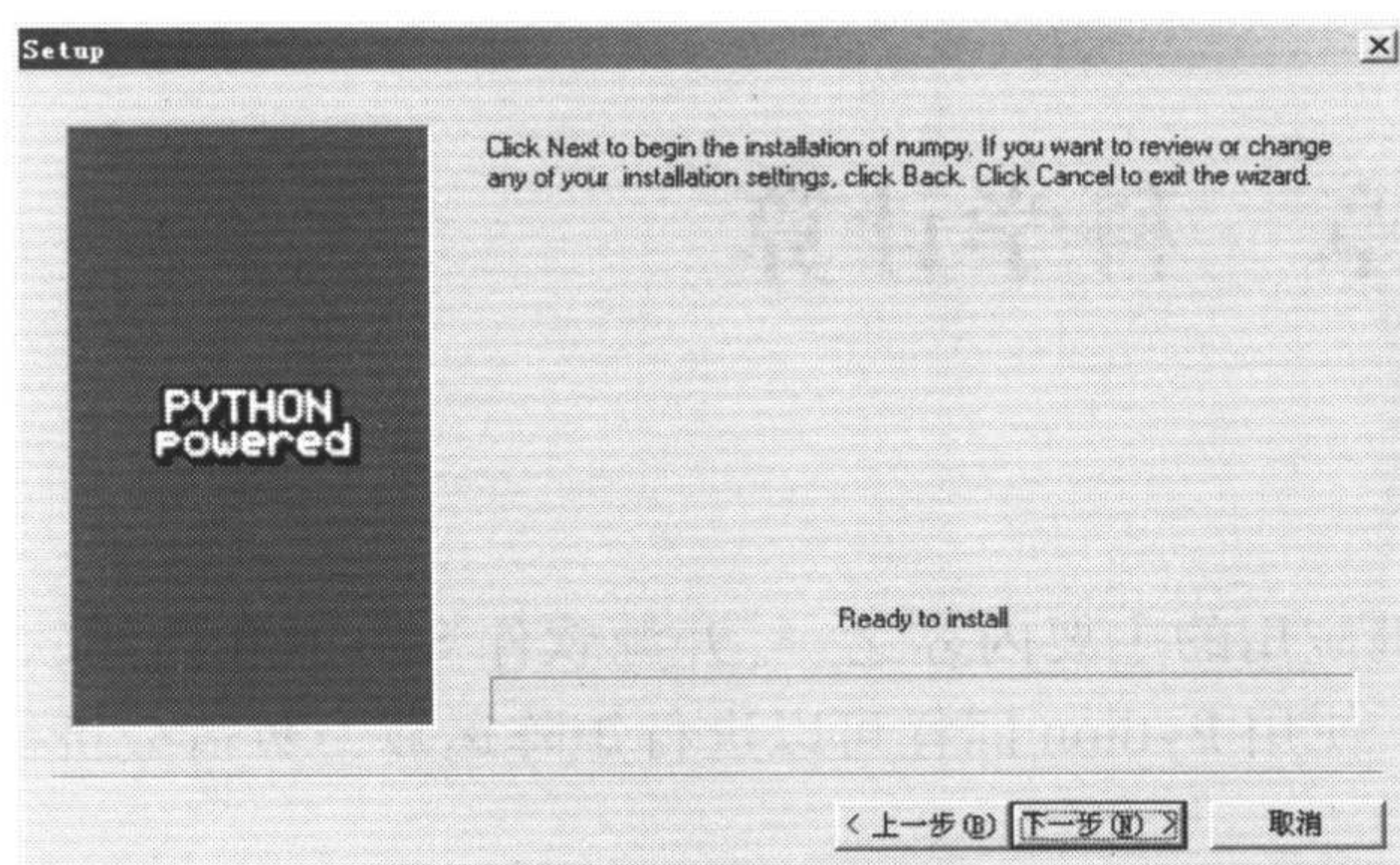


图 21-3 确认安装

2. 安装 SciPy

以 Python 2.5 为例，在 Windows 下安装 SciPy 的步骤如下所示。

(1) 从 SciPy 官方网站 <http://www.scipy.org> 下载 SciPy 在 Windows 下的安装包 `scipy-0.5.2.win32-py2.5.exe`。

(2) 双击运行安装程序，如图 21-4 所示。

(3) 单击【下一步】按钮，将出现如图 21-5 所示的安装路径界面。

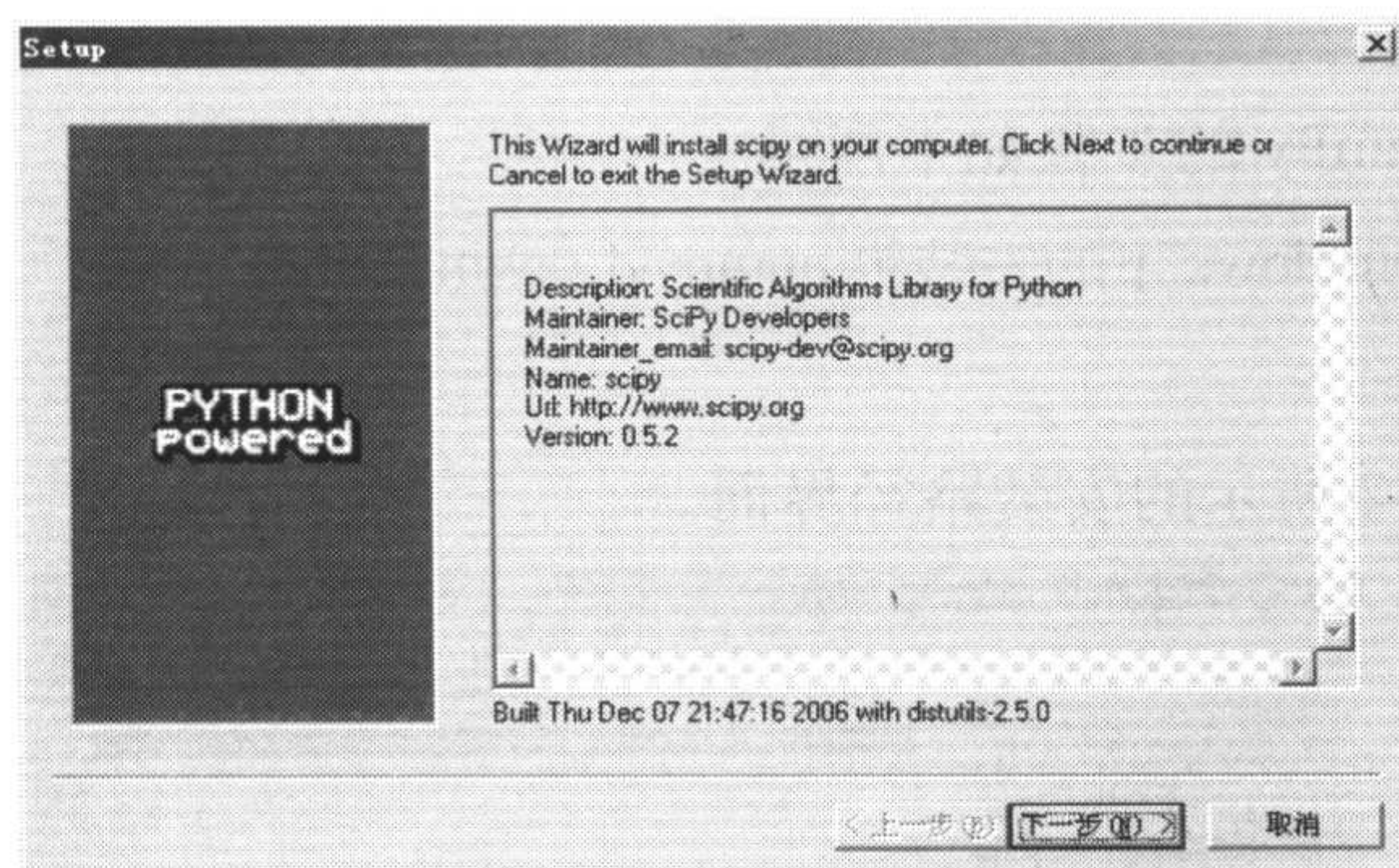


图 21-4 SciPy 安装程序

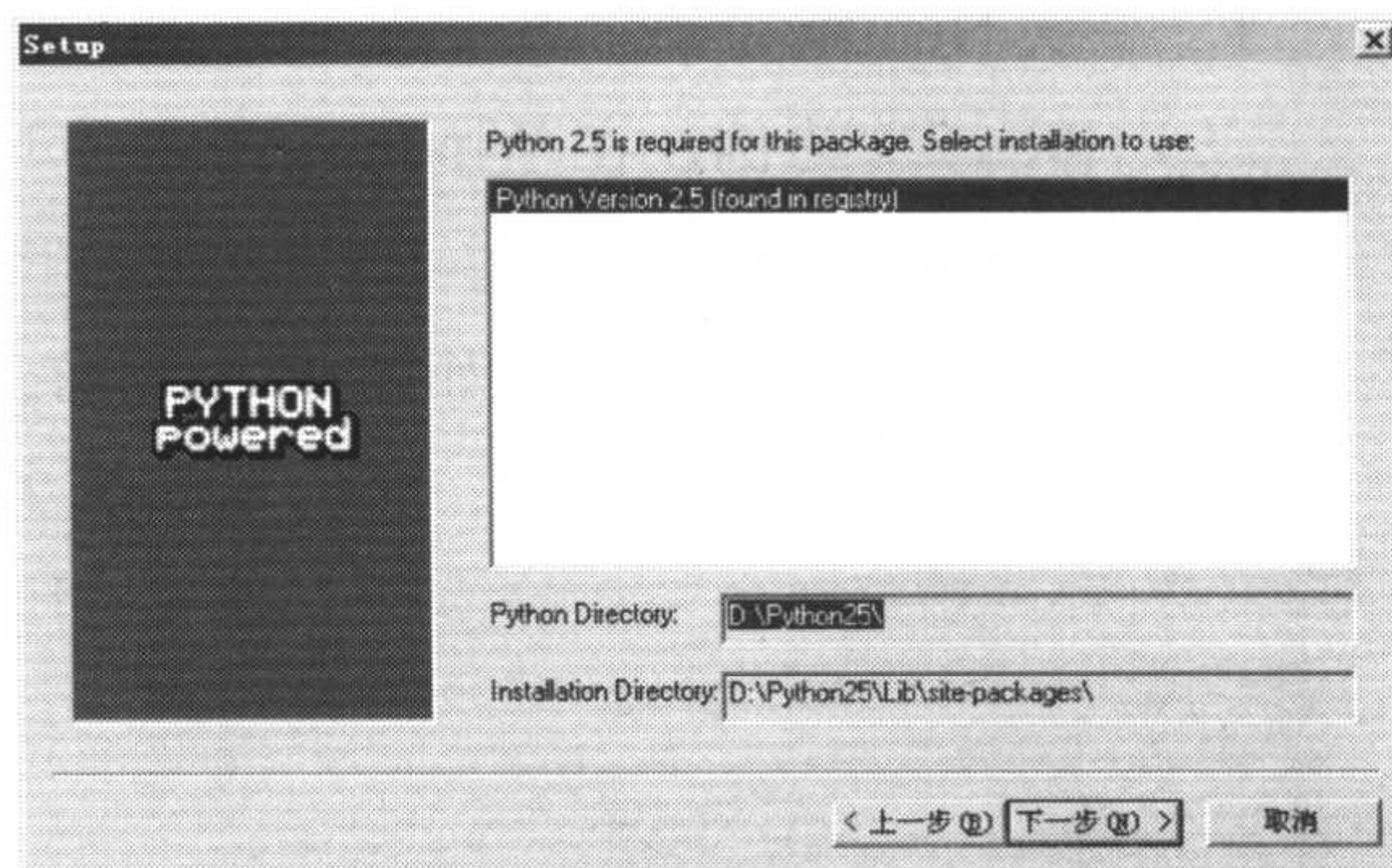


图 21-5 安装路径

(4) 单击【下一步】按钮，进入安装确认界面，如图 21-6 所示。单击【下一步】按钮完成 SciPy 安装。

需要注意的是，如果安装的 NumPy 比 SciPy 发布日期晚，则使用 SciPy 时将出现警告。如果使用 `scipy-0.5.2.win32-py2.5.exe`，最好使用 NumPy 的 1.0 版或者 NumPy 的 1.0.1 版。

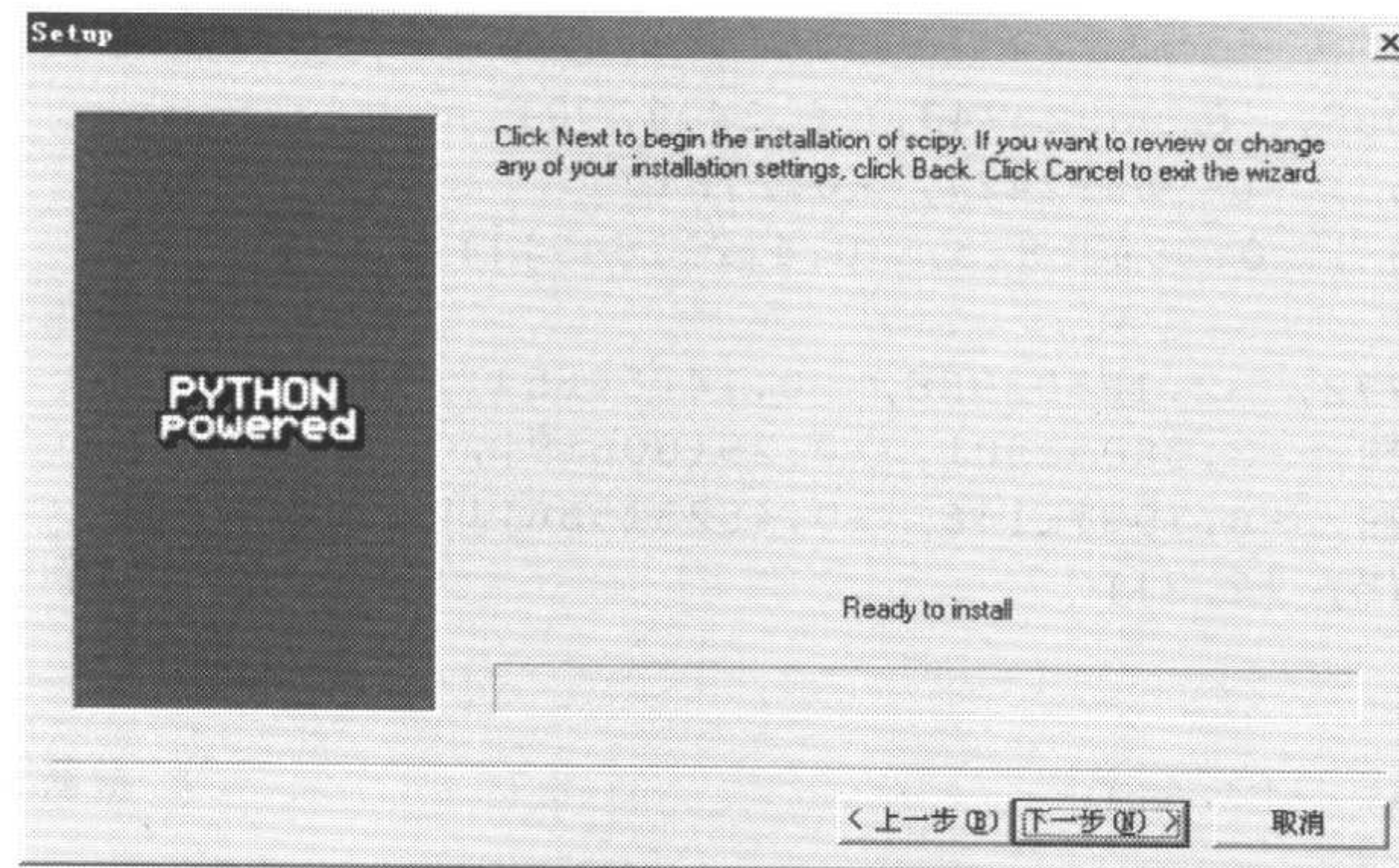


图 21-6 确认安装

21.1.2 NumPy 简介

NumPy 提供了 Python 没有提供的数组对象。使用 NumPy 的数组对象可以创建类似于 C 语言中的数组。使用 NumPy 时应首先导入 NumPy 模块，如下所示，在 Python 交互式命令行中使用 NumPy 创建数组，并对数组进行简单的运算。

```
>>> import numpy                                     # 导入 NumPy 模块
>>> a = numpy.array((1,2,3,4,5))                     # 生成一个数组对象
>>> print a                                           # 打印数组 a
[1 2 3 4 5]
>>> b = numpy.array([[1,2,3],[4,5,6],[7,8,9]])       # 生成数组对象 b
>>> print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> c = b + b                                         # 数组对象的加法运算
>>> print c
[[ 2  4  6]
 [ 8 10 12]
[14 16 18]]
>>> d = c * 2                                         # 数组对象的乘法运算
>>> print d
[[ 4  8 12]
[16 20 24]
[28 32 36]]
>>> e = d / c                                         # 数组对象的除法运算
>>> print e
[[2 2 2]
 [2 2 2]
 [2 2 2]]
>>> print b * e                                       # 数组相乘
[[ 2  4  6]
 [ 8 10 12]]
```



```
[14 16 18]]
>>> numpy.sin(b)
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ],
       [ 0.6569866 ,  0.98935825,  0.41211849]])
>>> numpy.tan(b)
array([[ 1.55740772, -2.18503986, -0.14254654],
       [ 1.15782128, -3.38051501, -0.29100619],
       [ 0.87144798, -6.79971146, -0.45231566]])
>>> numpy.resize(b, [2,2])
array([[1, 2],
       [3, 4]])
>>> numpy.resize(b, [3,4])
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 1, 2, 3]])
>>> numpy.sum(b)
45
>>> zero = numpy.zeros((4,4))
>>> print zero
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
>>> one = numpy.ones((4,4))
>>> print one
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

求正弦

求正切

重新调整 b 的大小, 生成新数组

重新调整 b 的大小

求 b 中所有元素的和

生成一个 4×4 的全为 0 的数组

生成一个 4×4 的全为 1 的数组

21.1.3 SciPy 简介

SciPy 模块依赖于 NumPy, 但是 SciPy 提供了更多的数学工具。使用 SciPy 不仅可以进行矩阵运算, 还可以求解线性方程组、进行积分运算、优化等。SciPy 的功能非常接近 Matlab。如下所示代码在 Python 交互式命令行中使用 SciPy。

```
>>> import scipy
>>> a = scipy.mat('[1 2 3; 4 5 6; 7 8 9]')
>>> a
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
>>> a * 2
matrix([[ 2,  4,  6],
        [ 8, 10, 12],
        [14, 16, 18]])
>>> 0.5 * a
matrix([[ 0.5,  1. ,  1.5],
```

导入 scipy 模块

生成一个矩阵

矩阵乘以 2

0.5 乘以矩阵 a

```

        [ 2. ,  2.5,  3. ],
        [ 3.5,  4. ,  4.5]])
>>> b = scipy.mat('[1;2;3]')
>>> print b
[[1]
 [2]
 [3]]
>>> a * b
matrix([[14],
        [32],
        [50]])
>>> scipy.sin(a)
matrix([[ 0.84147098,  0.90929743,  0.14112001],
        [-0.7568025 , -0.95892427, -0.2794155 ],
        [ 0.6569866 ,  0.98935825,  0.41211849]])
>>> scipy.tan(a)
matrix([[ 1.55740772, -2.18503986, -0.14254654],
        [ 1.15782128, -3.38051501, -0.29100619],
        [ 0.87144798, -6.79971146, -0.45231566]])
>>> scipy.resize(a, (2,3))
array([[1, 2, 3],
       [4, 5, 6]])
>>> scipy.resize(a, (4,4))
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 1, 2, 3],
       [4, 5, 6, 7]])
>>> v = scipy.vander((1,6))
>>> v
array([[1, 1],
       [6, 1]])
>>> scipy.diff(v)
array([[ 0],
       [-5]])
>>> from scipy import integrate
>>> integrate.quad(lambda x: 2*x, 0, 6)
(36.0, 3.9968028886505635e-013)
>>> integrate.quad(lambda x: 1/(1 + x ** 2), 0, 1)
(0.78539816339744839, 8.7196712450215814e-015)
```

生成矩阵
输出矩阵

矩阵相乘

求正弦

求正切

重新调整矩阵大小 2×3

重新调整矩阵大小 4×4

生成 Vandermonde 矩阵

差分运算

导入 integrate 模块

求积分 $\int_0^6 2x dx$

求积分 $\int_0^1 \frac{dx}{1+x^2}$

21.2 矩阵运算和解线性方程组

使用 SciPy 可以完成矩阵分解、线性方程组求解和多项式求根等数学运算。SciPy 中提供的函数名与 Matlab 中的函数名大部分都相同，用法也差不多。熟悉 Matlab 的用户可以很快熟悉 SciPy。

21.2.1 矩阵运算

除了基本的矩阵乘除运算以外，还可以使用 SciPy 做矩阵分解运算、求逆运算等。SciPy 中的 linalg 模块提供了和线性代数相关的函数，可以用于对矩阵进行运算。常用的 linalg 模块中常用的函数如表 21-1 所示。

表 21-1 linalg 模块中的常用函数

函 数	描 述	函 数	描 述
inv	求解矩阵的逆	orth	求解矩阵的标准正交基
det	求解方阵的行列式	lu	矩阵的 LU 分解
norm	求解向量的模	qr	矩阵的 QR 分解
lstsq	求最小二乘法解	cholesky	矩阵的 Cholesky 分解
eig	求解特征值和特征向量		

如下代码在 Python 的交互式命令行中使用 SciPy 中的 linalg 模块对矩阵进行运算。

```
>>> import scipy # 导入 SciPy 模块
>>> from scipy import linalg # 导入 linalg 模块
>>> a = scipy.mat('[1 2 3; 2 2 1; 3 4 3]') # 生成矩阵
>>> b = linalg.inv(a) # 求 a 的逆矩阵
>>> print b
[[ 1.  3. -2. ]
 [-1.5 -3.  2.5]
 [ 1.  1. -1. ]]
>>> a * b # 求 axb, 有误差, 应为对角矩阵
matrix([[ 1.00000000e+00, -4.44089210e-16, -4.44089210e-16],
 [ 0.00000000e+00, 1.00000000e+00, -2.22044605e-16],
 [-4.44089210e-16, 0.00000000e+00, 1.00000000e+00]])
>>> linalg.det(a) # 求 a 的行列式值, 有误差, 应为 2
1.9999999999999996
>>> a = scipy.mat('[1 2 3]')
>>> linalg.norm(a) # 求 a 的模
3.74165738677

#如下所示步骤求超定方程
{
x1 - x2 = 1
-x1 + x2 = 2
2x1 - 2x2 = 3
-3x1 + x2 = 4
} 的最小二乘解

>>> a = scipy.mat('[1 -1; -1 1; 2 -2; -3 1]') # 建立系数矩阵
>>> a # 检验系数矩阵
matrix([[ 1, -1],
 [-1,  1],
 [ 2, -2],
```

第21章 科学计算

```

        [-3, 1]])
>>> b = scipy.mat('[1;2;3;4]')
>>> b
matrix([[1],
        [2],
        [3],
        [4]])
>>> x,y,z,w = linalg.lstsq(a,b)
>>> x
array([[ -2.41666667],
       [ -3.25      ]])
>>> y
array([ 9.83333333])
>>> z
2
>>> w
array([ 4.56605495,  1.07291295])
>>> a = scipy.mat('[-1 1 0; -4 3 0; 1 0 2]')
>>> print a
[[-1  1  0]
 [-4  3  0]
 [ 1  0  2]]
>>> x,y = linalg.eig(a)
>>> x
array([ 2.      +0.j,  0.99999998+0.j,  1.00000002+0.j])
>>> y
array([[ 0.      ,  0.40824829, -0.40824829],
       [ 0.      ,  0.81649658, -0.81649658],
       [ 1.      , -0.40824829,  0.40824829]])
>>> a = scipy.mat('[1 2 3; 0 1 2; 2 4 1]')
>>> print a
[[1 2 3]
 [0 1 2]
 [2 4 1]]
>>> x,y,z = linalg.lu(a)
>>> x
array([[ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [ 1.,  0.,  0.]])
>>> y
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.5,  0.,  1.]])
>>> z
array([[ 2.,  4.,  1.],
       [ 0.,  1.,  2.],
       [ 0.,  0.,  2.5]])
>>> a = scipy.mat('[16 4 8; 4 5 -4; 8 -4 22]')
>>> print a

```

建立常数项向量

求最小二乘法解
x 为解

求 a 的特征值
a 的特征值, 有误差, 应为[2, 1, 1]
a 的特征向量

求 a 的 LU 分解

L 矩阵

U 矩阵


```
[[16  4  8]
 [ 4  5 -4]
 [ 8 -4 22]]
>>> linalg.cholesky(a)                                     # 求 a 的 Cholesky 分解
array([[ 4.,  1.,  2.],
       [ 0.,  2., -3.],
       [ 0.,  0.,  3.]])
```

21.2.2 解线性方程组

SciPy 中的 linalg 模块提供了基本的解线性方程组的函数，只要给出方程的系数矩阵和常数项向量，就可以获得方程组的解。对于非线性方程组可以采用数值解法。linalg 模块中的 solve 函数可以用于解线性方程组，有如下所示的方程组。

$$\begin{cases} 2x_1 + x_2 - 5x_3 + x_4 = 8 \\ x_1 - 3x_2 - 6x_4 = 9 \\ 2x_2 - x_3 + 2x_4 = -5 \\ x_1 + 4x_2 - 7x_3 + 6x_4 = 0 \end{cases}$$

其系数矩阵为： $\begin{bmatrix} 2 & 1 & -5 & 1 \\ 1 & -3 & 0 & -6 \\ 0 & 2 & -1 & 2 \\ 1 & 4 & -7 & 6 \end{bmatrix}$ ，其常数项向量为： $\begin{bmatrix} 8 \\ 9 \\ -5 \\ 0 \end{bmatrix}$ 。

如下所示的代码在 Python 交互式命令行中，使用 linalg 模块中的 solve 函数解上述线性方程组。

```
>>> import scipy
>>> from scipy import linalg
>>> a = scipy.mat('[2 1 -5 1;1 -3 0 -6;0 2 -1 2;1 4 -7 6]')
>>> print a
[[ 2  1 -5  1]
 [ 1 -3  0 -6]
 [ 0  2 -1  2]
 [ 1  4 -7  6]]
>>> b = scipy.mat('[8;9;-5;0]')
>>> print b
[[ 8]
 [ 9]
 [-5]
 [ 0]]
>>> linalg.solve(a,b)
array([[ 3.],
       [-4.],
       [-1.],
       [ 1.]])
```


21.3 使用 Matplotlib 绘制函数图形

Matplotlib 是绘制 2D 图形的 Python 的一种模块,它依赖于 NumPy 和 Tkinter。Matplotlib 中大部分函数都和 Matlab 中的函数名相同,熟悉 Matlab 的用户可以很快地掌握 Matplotlib。Matplotlib 可以绘制多种形式的图形包括普通的线图、直方图、饼图、散点图以及误差线图。

21.3.1 安装 Matplotlib

Matplotlib 提供了 Windows 下的安装程序,以 Python 2.5 为例,其安装步骤如下所示。

(1) 从 Matplotlib 官方网站 <http://matplotlib.sourceforge.net> 下载 matplotlib-0.90.0.win32-py2.5.exe 安装程序。

(2) 双击运行安装程序,如图 21-7 所示。

(3) 单击【下一步】按钮,选择安装路径界面,如图 21-8 所示。

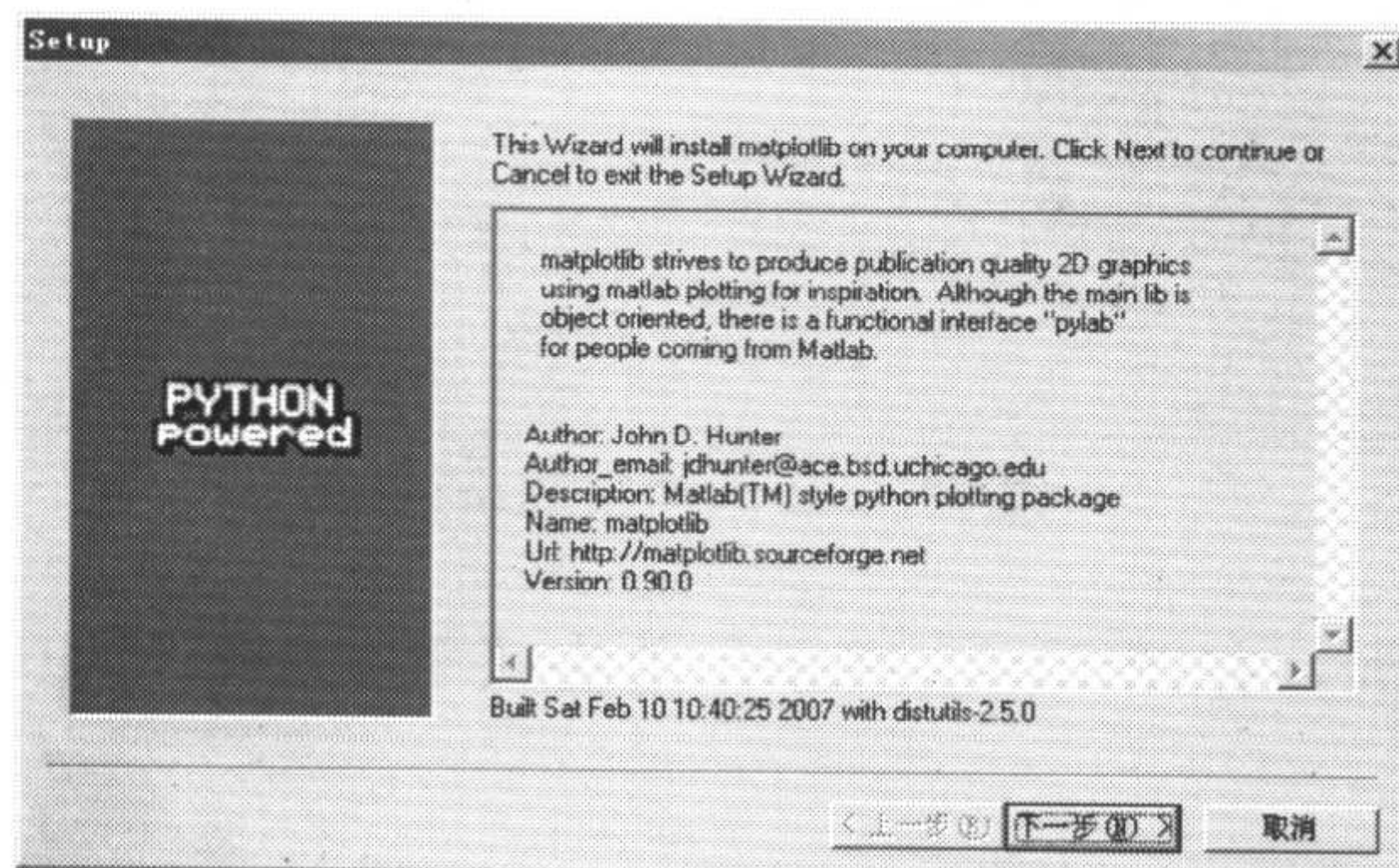


图 21-7 Matplotlib 安装程序

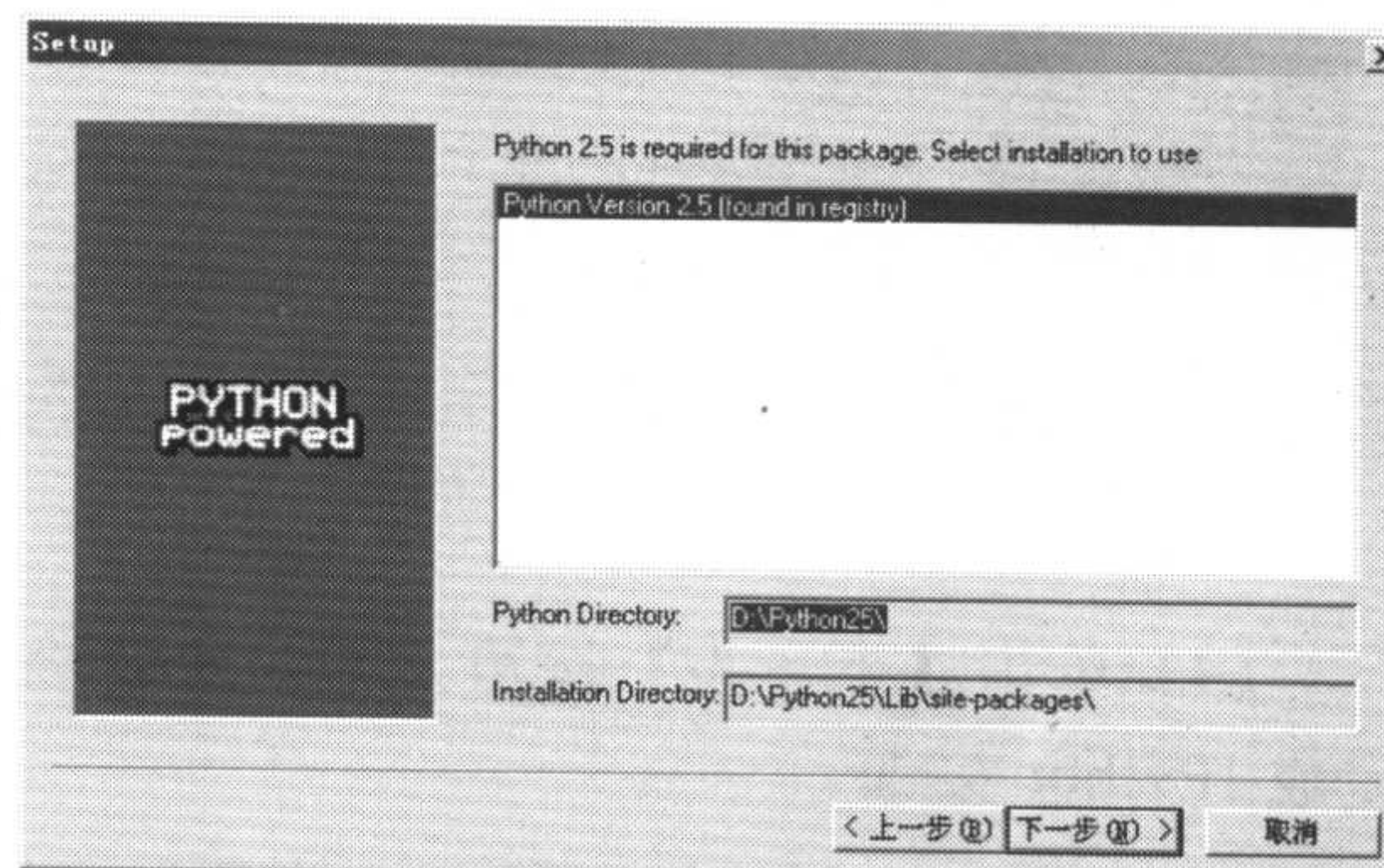


图 21-8 安装路径

(4) 单击【下一步】按钮,进入安装确认界面,如图 21-9 所示。单击【下一步】按钮,完成 Matplotlib 安装。

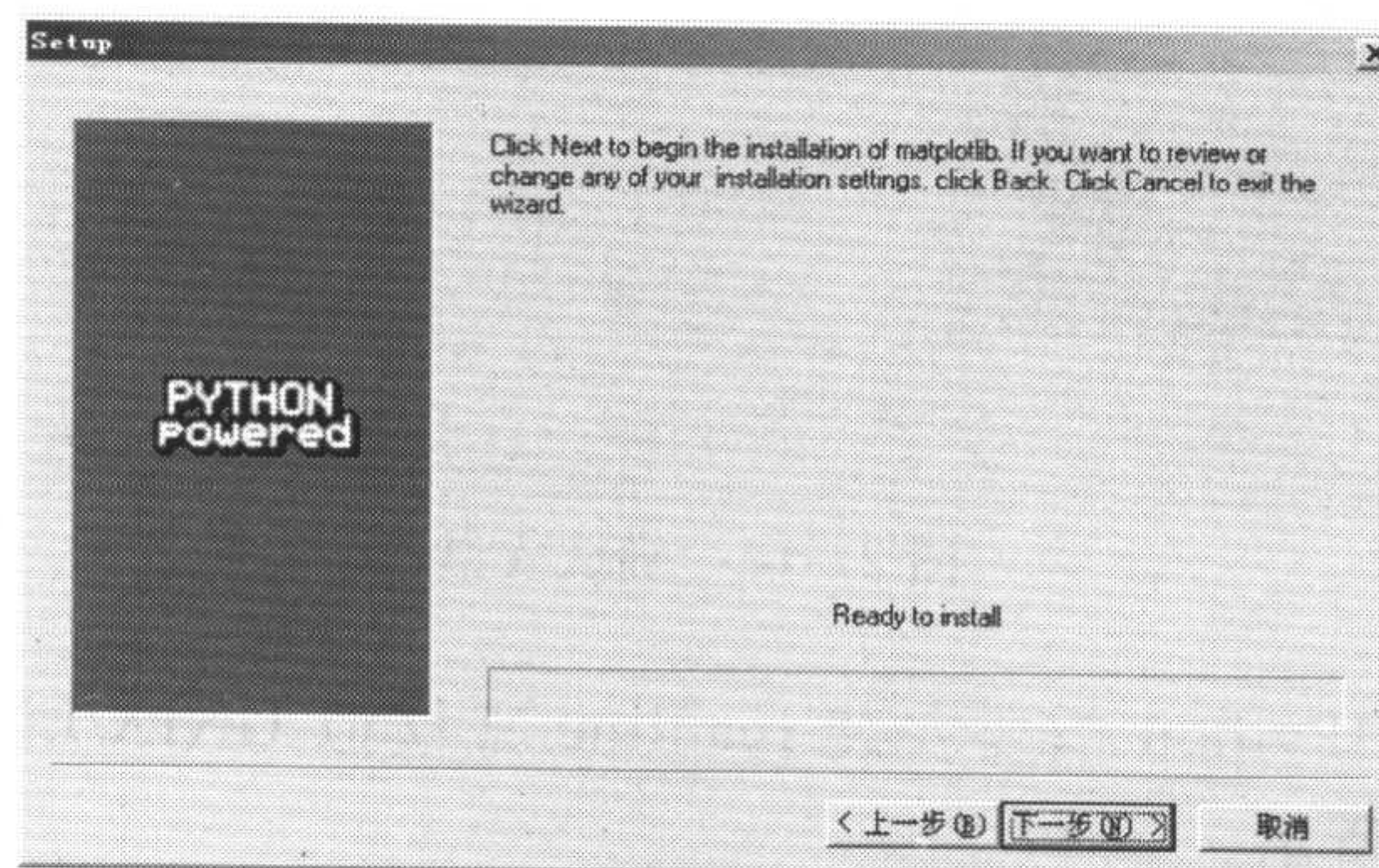


图 21-9 确认安装

为了能和 Matlab 一样在交互式命令行下绘制图形，需要安装 IPython。IPython 是更高级的 Python 交互式命令行，它提供了对 Matplotlib 的支持。以 Python 2.5 为例，在 Windows 下安装 IPython 步骤如下所示。

(1) 从 IPython 官方网站 <http://ipython.scipy.org> 下载 Windows 下的安装程序 `ipython-0.8.1.win32.exe`。

(2) 双击运行安装程序，如图 21-10 所示。

(3) 单击【下一步】按钮，IPython 将搜索本机所安装的 Python 版本，根据需要选择要使用的 Python 版本，如图 21-11 所示。

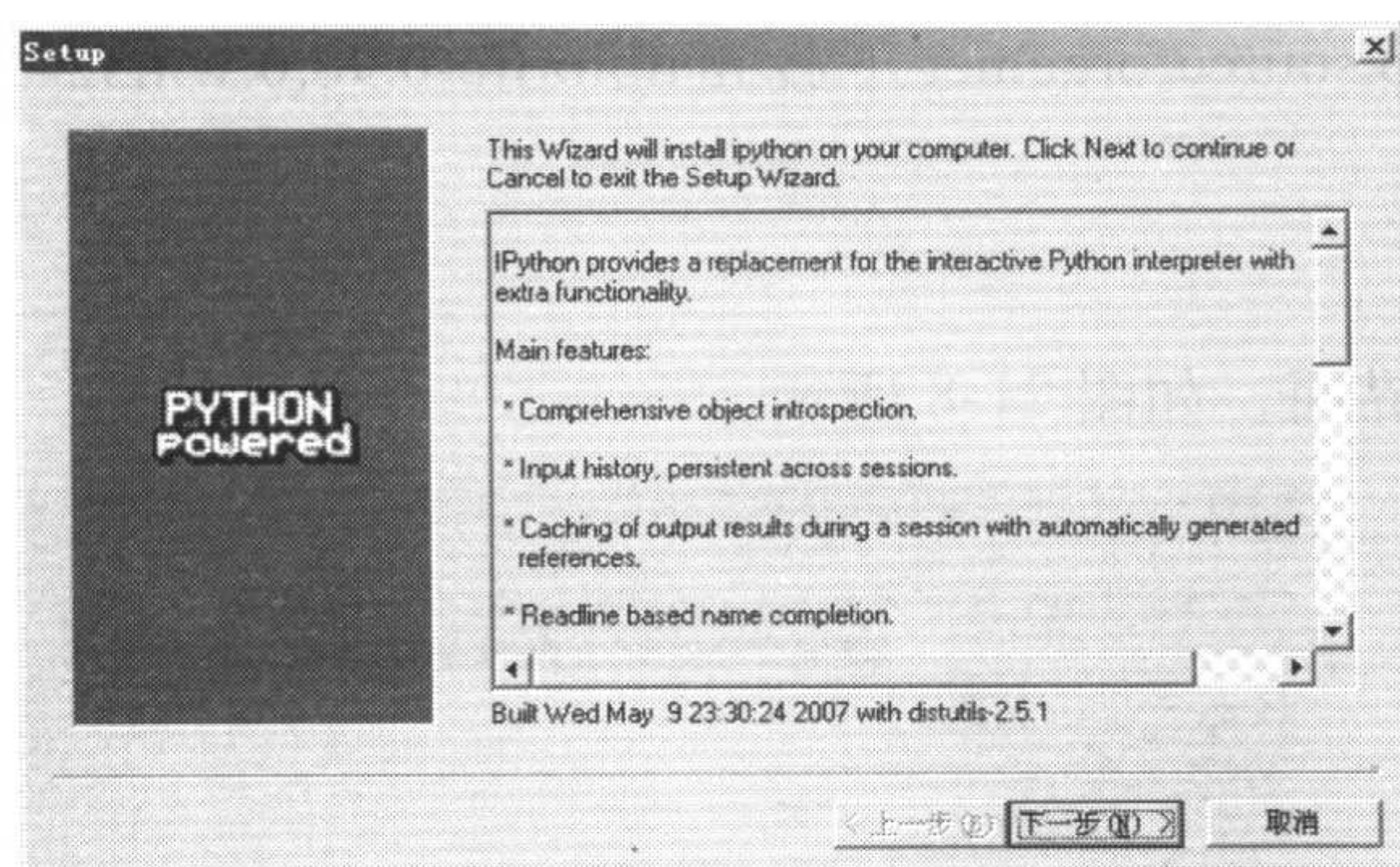


图 21-10 IPython 安装程序

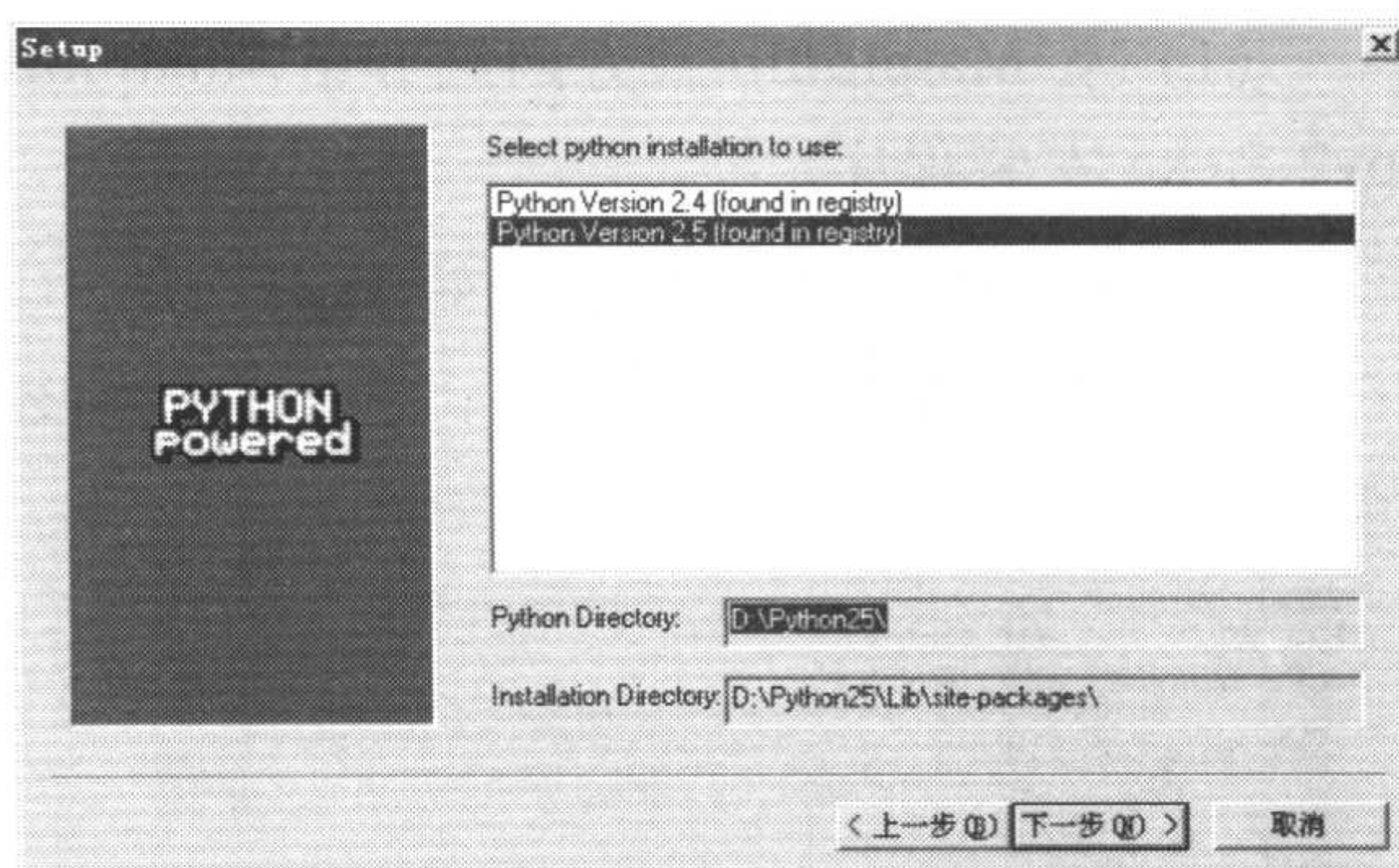


图 21-11 选择安装路径

(4) 单击【下一步】按钮，进入安装确认界面，如图 21-12 所示。单击【下一步】按钮，完成 IPython 安装。

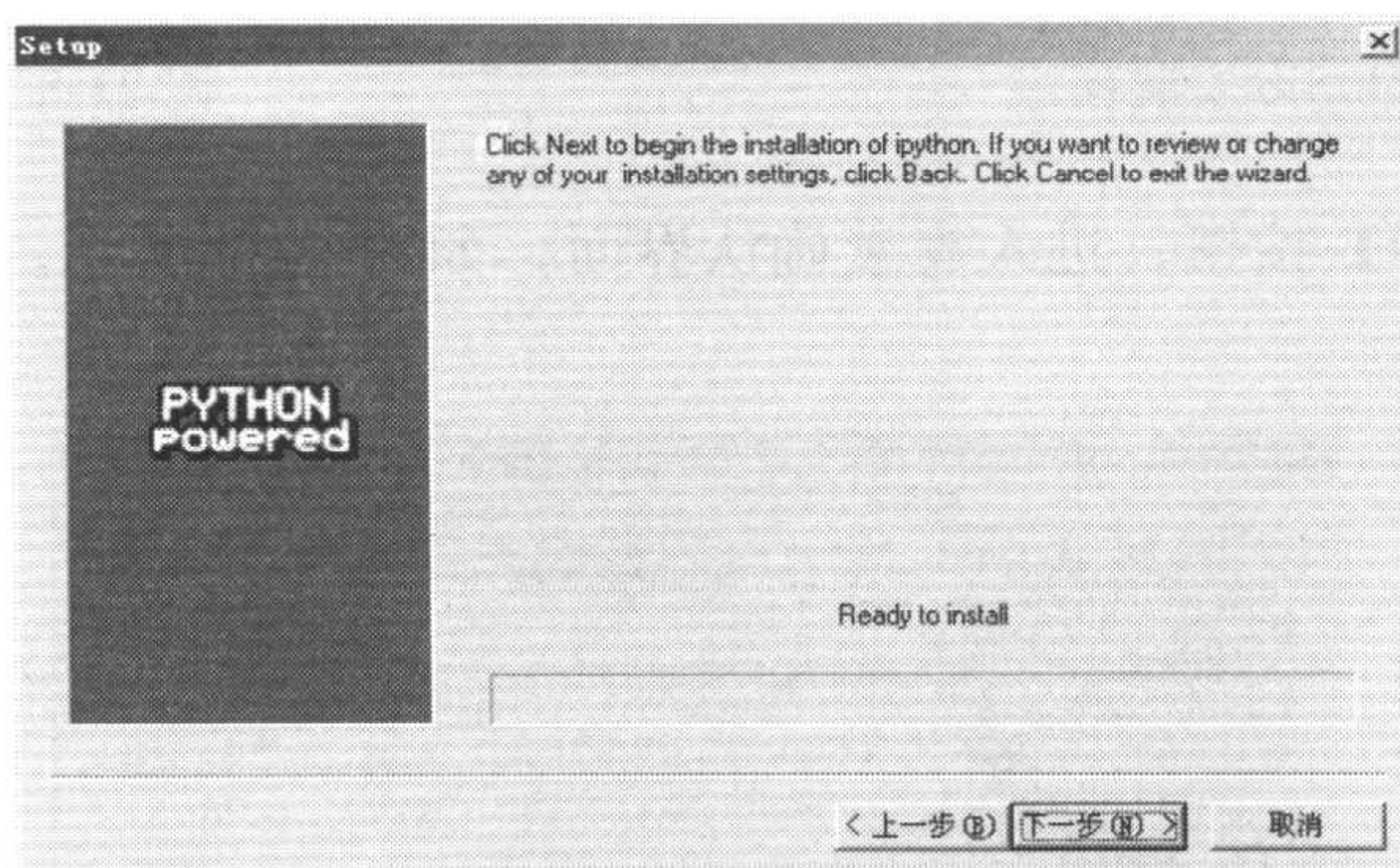


图 21-12 确认安装

在 Windows 环境下 IPython 需要安装 readline 来模拟 UNIX/Linux 下的 readline 功能。readline 的安装过程如下所示。

(1) 从 IPython 的官方网站 <http://ipython.scipy.org> 下载 readline 的安装程序 `pyreadline-`

1.3.win32.exe。

(2) 双击运行安装程序，如图 21-13 所示。

(3) 单击【下一步】按钮，readline 将搜索本机所安装的 Python 版本，根据需要选择要使用的 Python 版本，如图 21-14 所示。

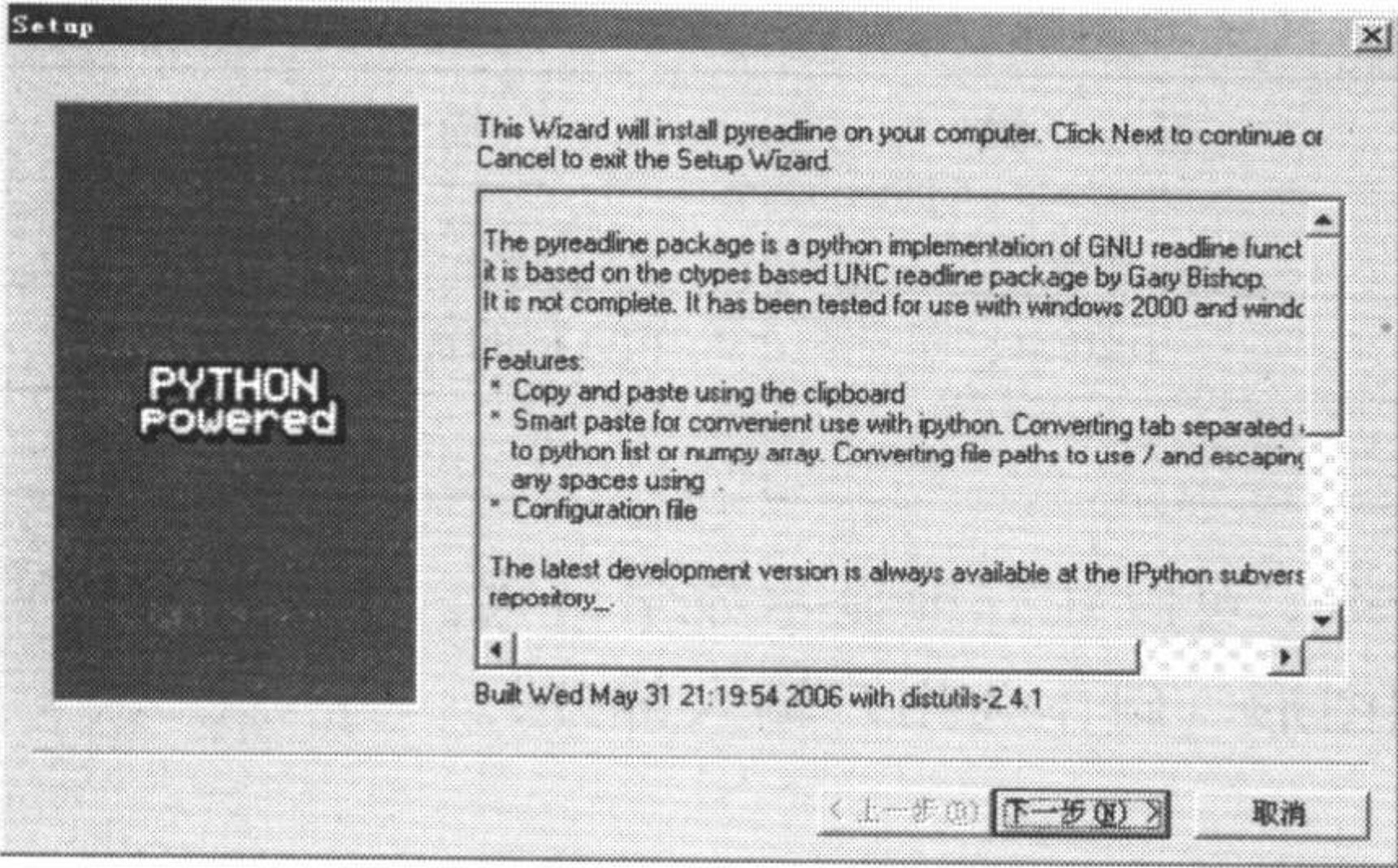


图 21-13 readline 安装程序

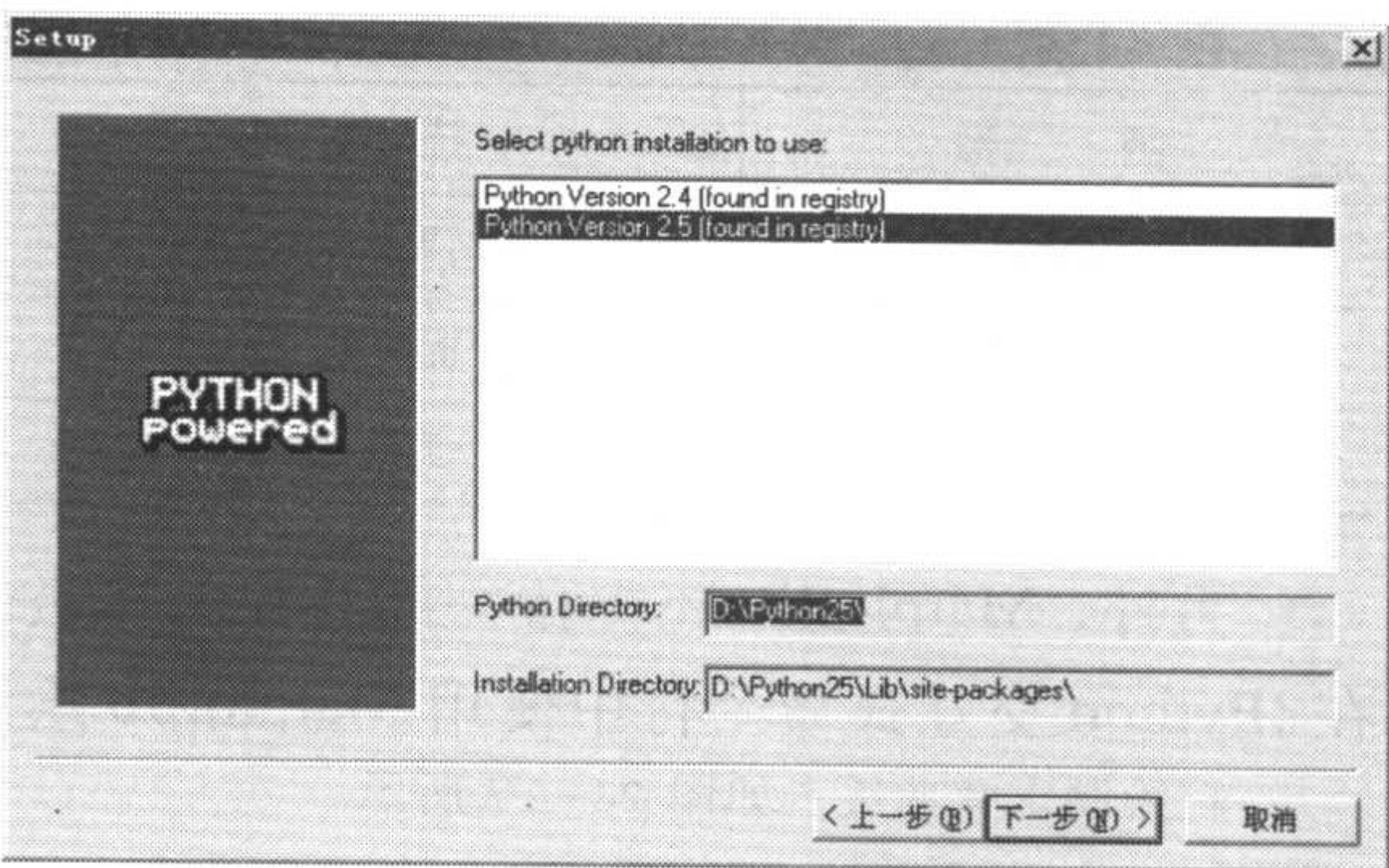


图 21-14 安装路径

(4) 单击【下一步】按钮，进入安装确认界面，如图 21-15 所示。单击【下一步】按钮，完成 readline 安装。

由于要在 IPython 下使用 Matplotlib 模块进行交互式地绘图，因此可以修改 IPython 的启动参数，直接进入 IPython 的 pylab 模式。单击菜单【开始】|【所有程序】|【IPython】命令，在弹出菜单中的【IPython】项右击，选择【属性】命令。在弹出的属性对话框中【快捷方式】标签下的目标文本框中添加“-pylab”参数，如图 21-16 所示。

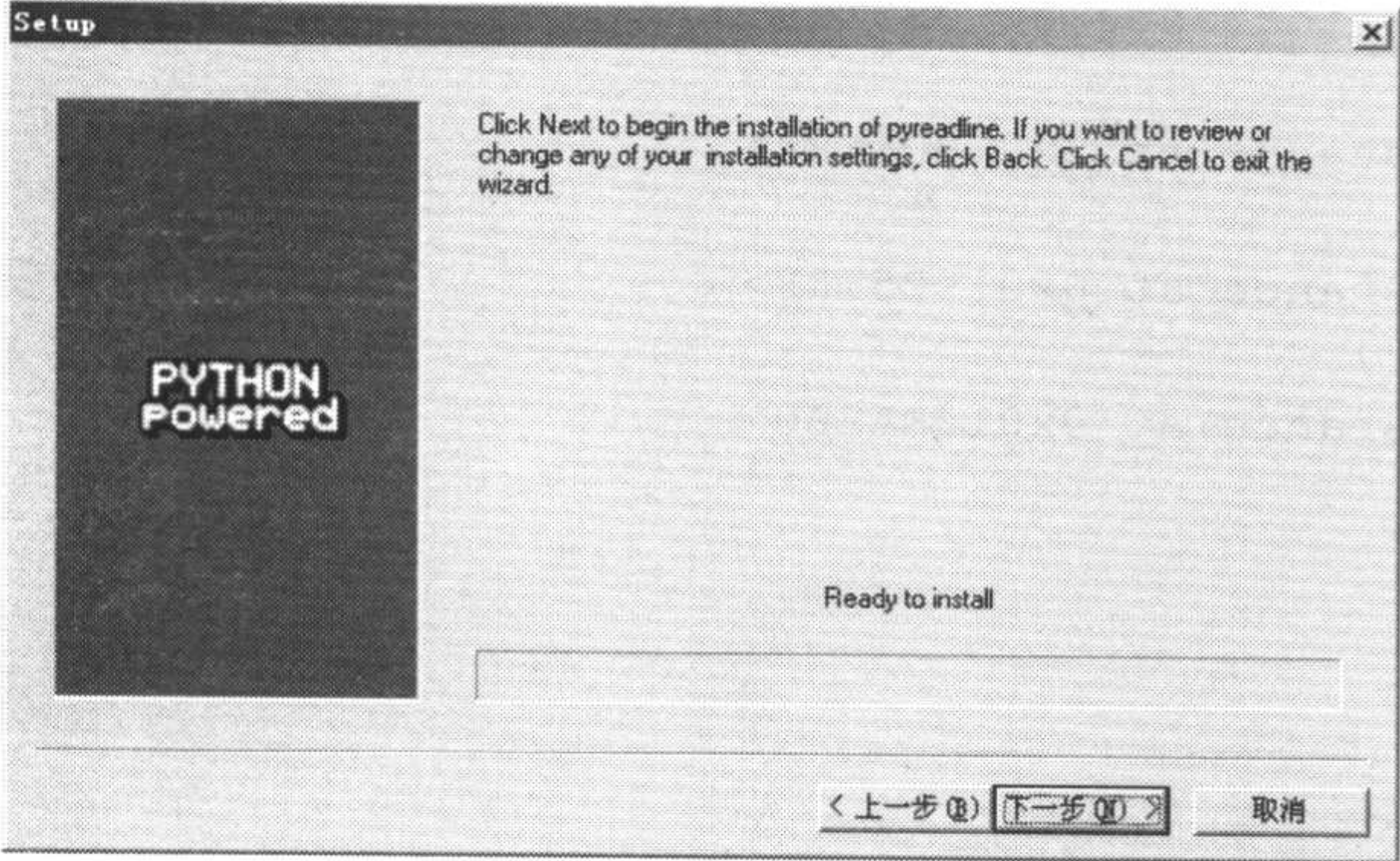


图 21-15 确认安装

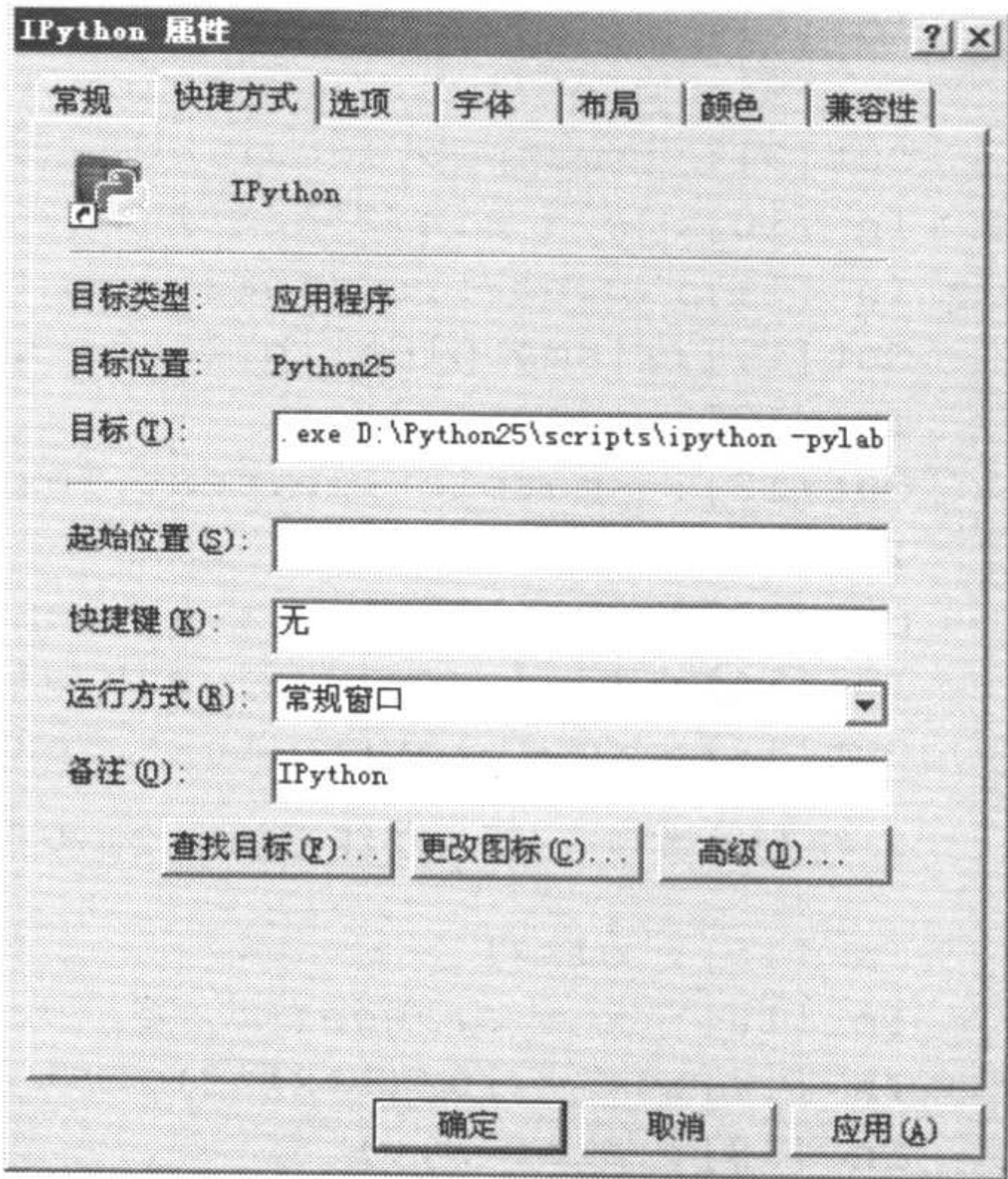


图 21-16 设置 IPython 启动参数

21.3.2 使用 Matplotlib 绘制图形

使用 Matplotlib 绘制图形主要使用其 plot 函数，plot 函数和 Matlab 中的 plot 函数用法相同。Matplotlib 还包含了一些用于设置 x、y 轴标签文本以及图形标题的函数，如表 21-2 所示。

表 21-2 Matplotlib 中设置文字的函数

函 数	描 述	函 数	描 述
xlabel	设置 x 轴标签	text	在指定坐标处输出文字
ylabel	设置 y 轴标签	figtext	在绘制的图形上添加文字
title	设置绘图标题		

另外，Matplotlib 还支持一部分 Tex 排版命令，可以较好地显示数学公式。如下所示代码在 IPython 交互式命令行中使用 Matplotlib 绘制图形（斜体为用户输入部分）。

绘制正弦曲线，如图 21-17 所示。

```
In [1]: from pylab import *
```

```
In [2]: t = arange(0.0, 2.0, 0.05)
```

```
In [3]: s = sin(2*pi*t)
```

```
In [4]: plot(t,s)
```

```
Out[4]: [<matplotlib.lines.Line2D instance at 0x0181EBC0>]
```

绘制余弦曲线，如图 21-18 所示。

```
In [5]: figure(2)
```

新建一个绘图窗口

```
Out[5]: <matplotlib.figure.Figure instance at 0x019D06E8>
```

```
In [6]: s = cos(2*pi*t)
```

```
In [7]: plot(t,s)
```

```
Out[7]: [<matplotlib.lines.Line2D instance at 0x019DB4E0>]
```

同时绘制正弦曲线和余弦曲线，如图 21-19 所示。

```
In [8]: clf()
```

清除图形

```
In [9]: plot(t,s,linestyle='-',marker='o')
```

```
Out[9]: [<matplotlib.lines.Line2D instance at 0x019EA440>]
```

```
In [10]: s = sin(2*pi*t)
```

```
In [11]: plot(t,s,linestyle='-.',marker='+')
```

```
Out[11]: [<matplotlib.lines.Line2D instance at 0x019F0D00>]
```

```
In [12]: xlabel('X')
```

```
Out[12]: <matplotlib.text.Text instance at 0x019DFEB8>
```

```
In [13]: ylabel('Y')
```

```
Out[13]: <matplotlib.text.Text instance at 0x019E57D8>
```

```
In [14]: title('sin(x) and sin(y)')
```

```
Out[14]: <matplotlib.text.Text instance at 0x019EA2B0>
```

绘制函数 $y = \frac{10}{1+x^2}$ 曲线，如图 21-20 所示。

```
In [15]: clf()
```

```
In [16]: x = arange(-5, 5, 0.1)
```

```
In [17]: y = 10/(1+x**2)
```

```
In [18]: plot(x,y)
```

```
Out[18]: [<matplotlib.lines.Line2D instance at 0x019FB468>]
```


第21章 科学计算

```

In [19]: xlabel('X')
Out[19]: <matplotlib.text.Text instance at 0x019F4080>
In [20]: ylabel('Y')
Out[20]: <matplotlib.text.Text instance at 0x019F4800>

```

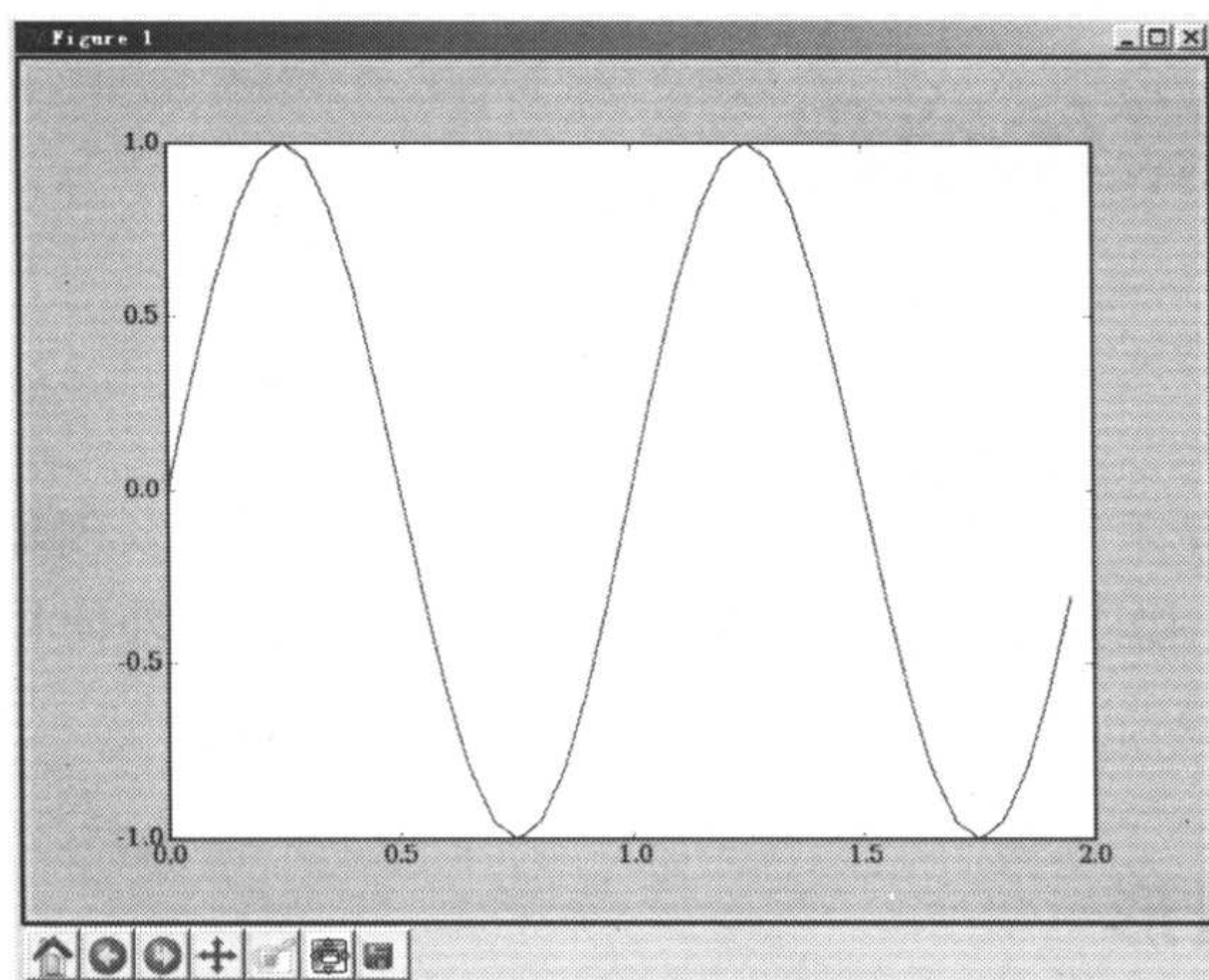


图 21-17 正弦曲线

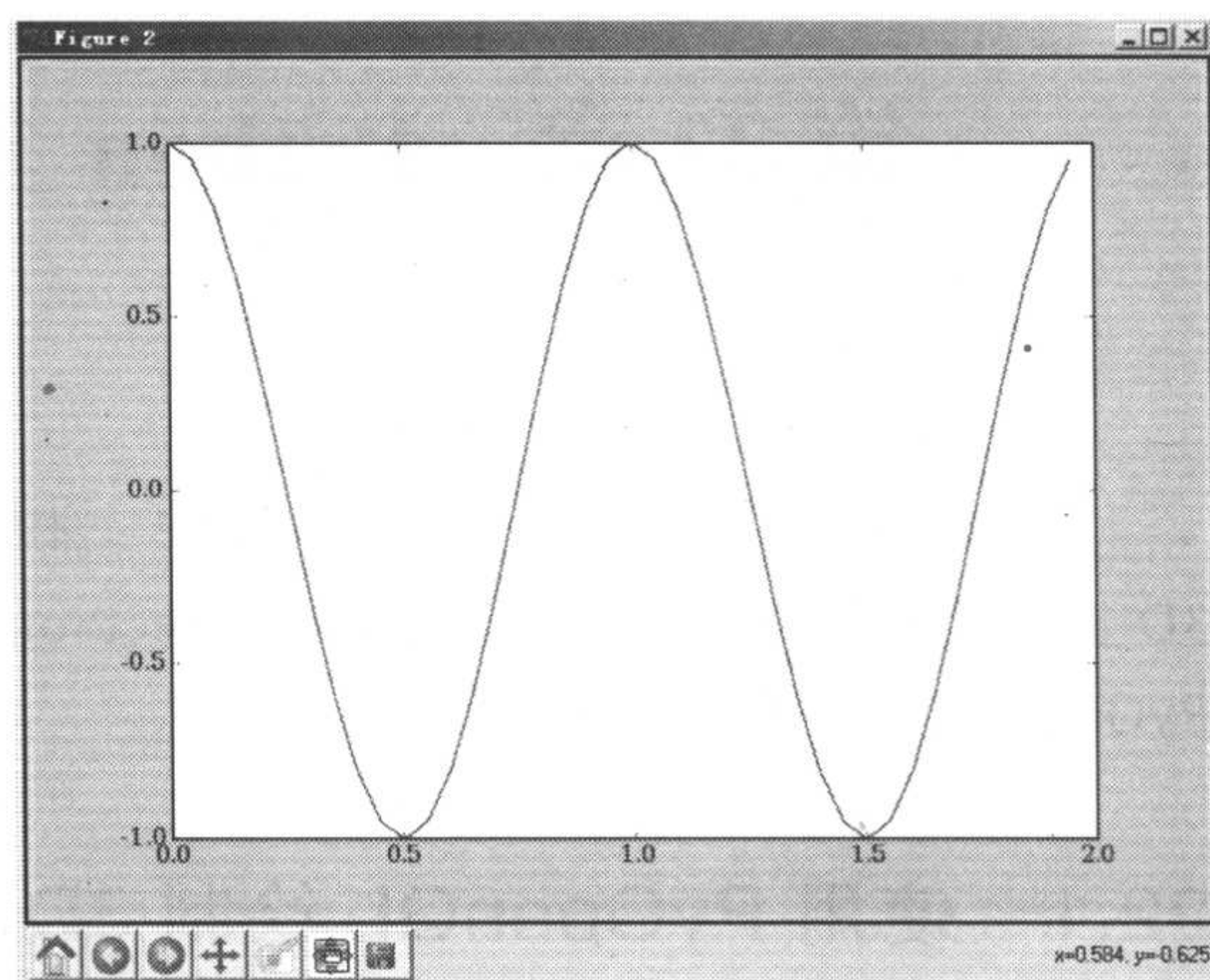


图 21-18 余弦曲线

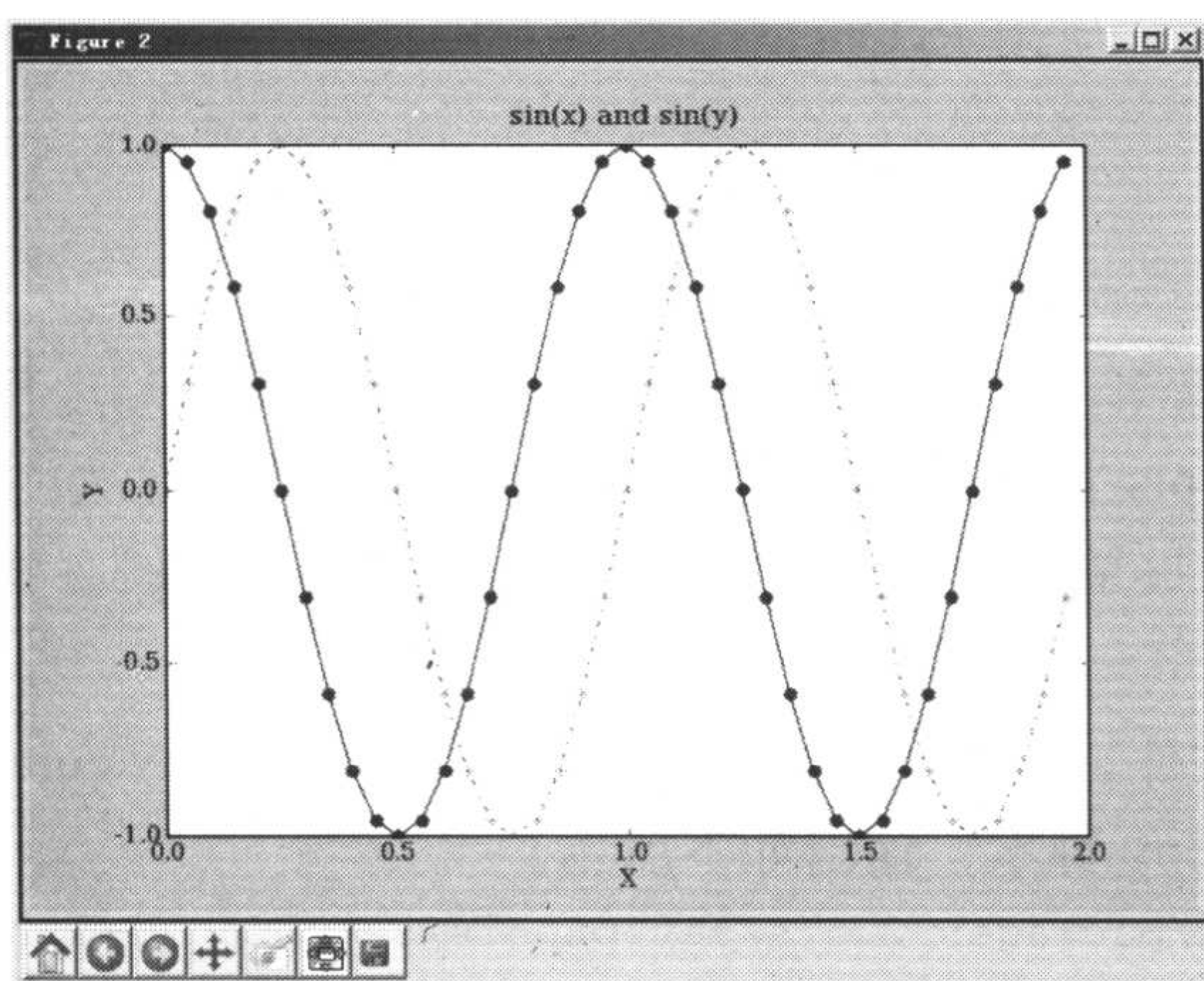
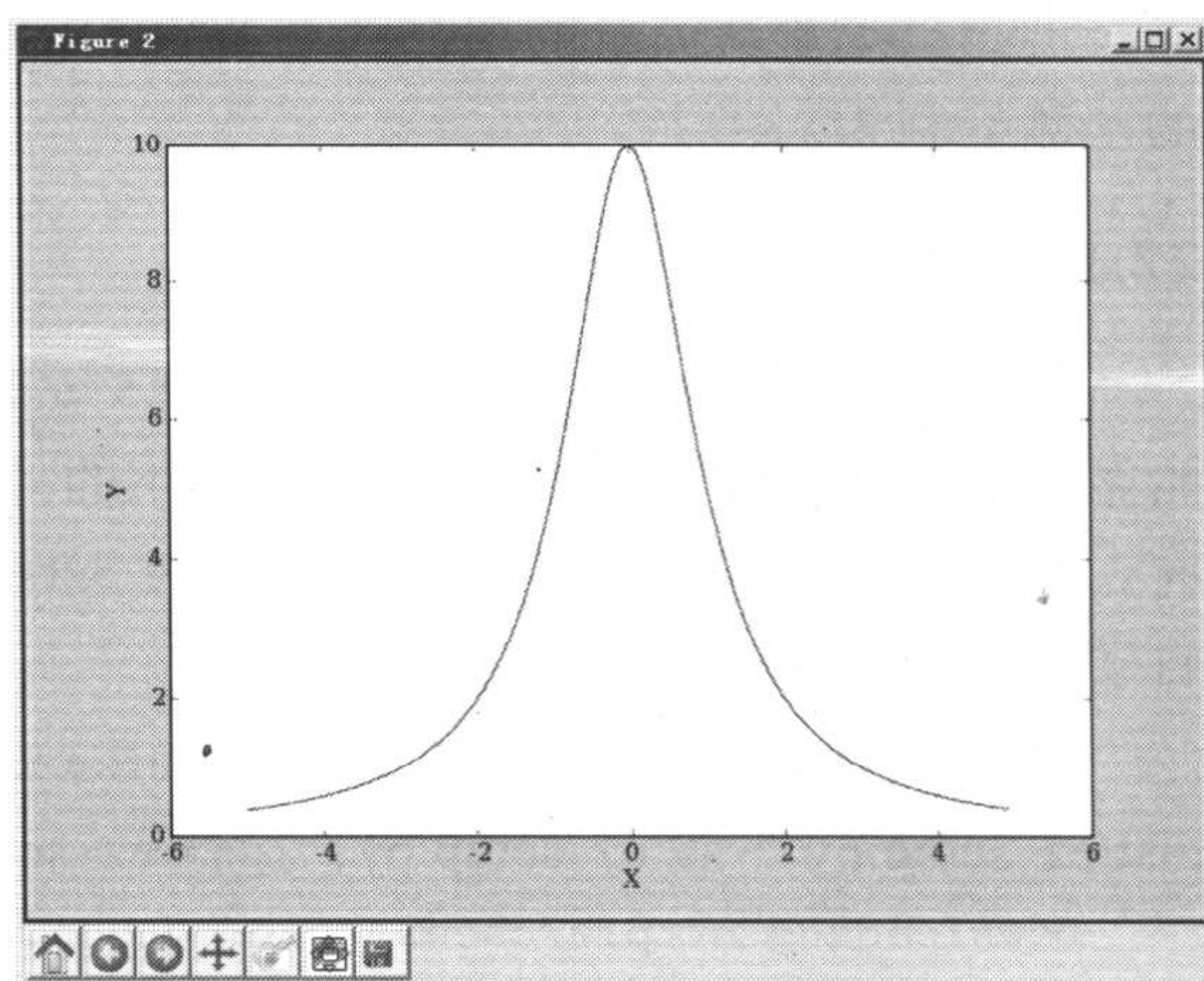


图 21-19 同时绘制正弦曲线和余弦曲线

图 21-20 函数 $y = \frac{10}{1+x^2}$ 曲线

第 22 章 Python 多媒体编程

使用 Python 可以进行创建 3D 图形、播放音乐等多媒体编程。使用 PyOpenGL 可以创建 3D 图形。在 Windows 下可以使用 DirectSound 和 WMPPlayer.OCX 来播音频文件。另外使用 PyGame 还可以编写游戏。

22.1 使用 PyOpenGL 绘制 3D 图形

PyOpenGL 模块是对 OpenGL 的封装，OpenGL (Open Graphics Library) 提供了不同的函数调用以绘制从简单的图形到复杂的 3D 图形。使用 PyOpenGL 模块可以使用 OpenGL 中的函数绘制 3D 图形。

22.1.1 安装 PyOpenGL

由于 PyOpenGL 没有提供 Python 2.5 安装程序，因此在 Windows 下安装 PyOpenGL 过程略微麻烦。首先查看 Python 安装目录下的 Scripts 目录中有没有 easy_install.exe 文件。如果有，则在 Windows 的命令行窗口中进入 Scripts 目录，输入“easy_install.exe PyOpenGL”，将自动安装 PyOpenGL。

如果 Scripts 目录中没有 easy_install.exe 文件，则需要从 http://peak.telecommunity.com/dist/ez_setup.py 下载 ez_setup.py，运行 ez_setup.py 脚本，输出如下所示。

```
Downloading http://cheeseshop.python.org/packages/2.5/s/setuptools/setuptools-0.6c5-py2.5.egg
Processing setuptools-0.6c5-py2.5.egg
Copying setuptools-0.6c5-py2.5.egg to d:\python25\lib\site-packages
Adding setuptools 0.6c5 to easy-install.pth file
Installing easy_install-script.py script to D:\Python25\Scripts
Installing easy_install.exe script to D:\Python25\Scripts
Installing easy_install-2.5-script.py script to D:\Python25\Scripts
Installing easy_install-2.5.exe script to D:\Python25\Scripts

Installed d:\python25\lib\site-packages\setuptools-0.6c5-py2.5.egg
Processing dependencies for setuptools==0.6c5
```

运行完 `ez_setup.py` 脚本后，将会在 Python 安装目录下的 `Scripts` 目录中创建 `easy_install-2.5.exe` 文件。在 Windows 的命令行窗口中进入 `Scripts` 目录，输入“`easy_install-2.5.exe PyOpenGL`”，输出如下所示。

```
D:\Python25\Scripts>easy_install-2.5.exe PyOpenGL
Searching for PyOpenGL
Reading http://cheeseshop.python.org/pypi/PyOpenGL/
Reading http://pyopengl.sourceforge.net/ctypes/
Reading http://sourceforge.net/project/showfiles.php?group_id=5988
Reading http://cheeseshop.python.org/pypi/PyOpenGL/3.0.0a5
Best match: PyOpenGL 3.0.0a6
Downloading http://downloads.sourceforge.net/pyopengl/PyOpenGL-3.0.0a6.zip?modti
me=1171556482&big_mirror=0
Processing PyOpenGL-3.0.0a6.zip
Running PyOpenGL-3.0.0a6\setup.py -q bdist_egg --dist-dir c:\docume~1\firefly\lo
Cals~1\temp\easy_install-phmvv3\PyOpenGL-3.0.0a6\egg-dist-tmp-gi2cti
Adding pyopengl 3.0.0a6 to easy-install.pth file

Installed d:\python25\lib\site-packages\pyopengl-3.0.0a6-py2.5.egg
Processing dependencies for PyOpenGL
```

安装好 `PyOpenGL` 以后，还需要检查 Windows 安装目录下 `system32` 目录中是否有 `opengl32.dll`、`glu32.dll` 和 `glut32.dll` 3 个 DLL 文件。如果没有，则可以到 <http://www.dll-files.com> 搜索下载。

22.1.2 使用 PyOpenGL 创建窗口

使用 `PyOpenGL` 和使用 `OpenGL` 创建程序的过程基本类似，在程序中首先对 `OpenGL` 进行初始化，设置相关的参数。在使用 `OpenGL` 的程序中首先调用 `glutInit` 函数，向其传递命令行参数。然后调用 `glutInitDisplayMode` 函数设置显示模式，调用 `glutCreateWindow` 函数创建窗口，调用 `glutDisplayFunc` 设置场景绘制函数。最后调用自定义的初始函数完成 `OpenGL` 的初始化，进入消息循环。如下所示的 `pyOpenGLWindow.py` 脚本仅创建了一个窗口。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLWindow.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:
    def __init__(self, width = 640, height = 480, title = 'PyOpenGL'):
        # 初始化
        glutInit(sys.argv)          # 传递命令行参数
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
        # 设置显示模式
        glutInitWindowSize(width, height)  # 设置窗口大小
        self.window = glutCreateWindow(title)  # 创建窗口
```



```

        glutDisplayFunc(self.Draw)
        self.InitGL(width, height)
    def Draw(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()
        glutSwapBuffers()
    def InitGL(self, width, height):
        glClearColor(0.0, 0.0, 0.0, 0.0)
        glClearDepth(1.0)
        glDepthFunc(GL_LESS)
        glEnable(GL_DEPTH_TEST)
        glShadeModel(GL_SMOOTH)
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)

        glMatrixMode(GL_MODELVIEW)
    def MainLoop(self):
        glutMainLoop()
window = OpenGLWindow()
window.MainLoop()

```

设置场景绘制函数
 # 调用 OpenGL 初始化函数
 # 绘制场景
 # 清除屏幕和深度缓存
 # 重置观察矩阵
 # 交换缓存
 # OpenGL 初始化函数
 # 设为黑色背景
 # 设置深度缓存
 # 设置深度测试类型
 # 允许深度测试
 # 启动平滑阴影
 # 设置观察矩阵
 # 重置观察矩阵
 # 计算屏幕高宽比
 # 设置观察矩阵
 # 进入消息循环
 # 创建窗口
 # 进入消息循环

运行 pyOpenGLWindow.py, 将创建如图 22-1 所示的窗口。

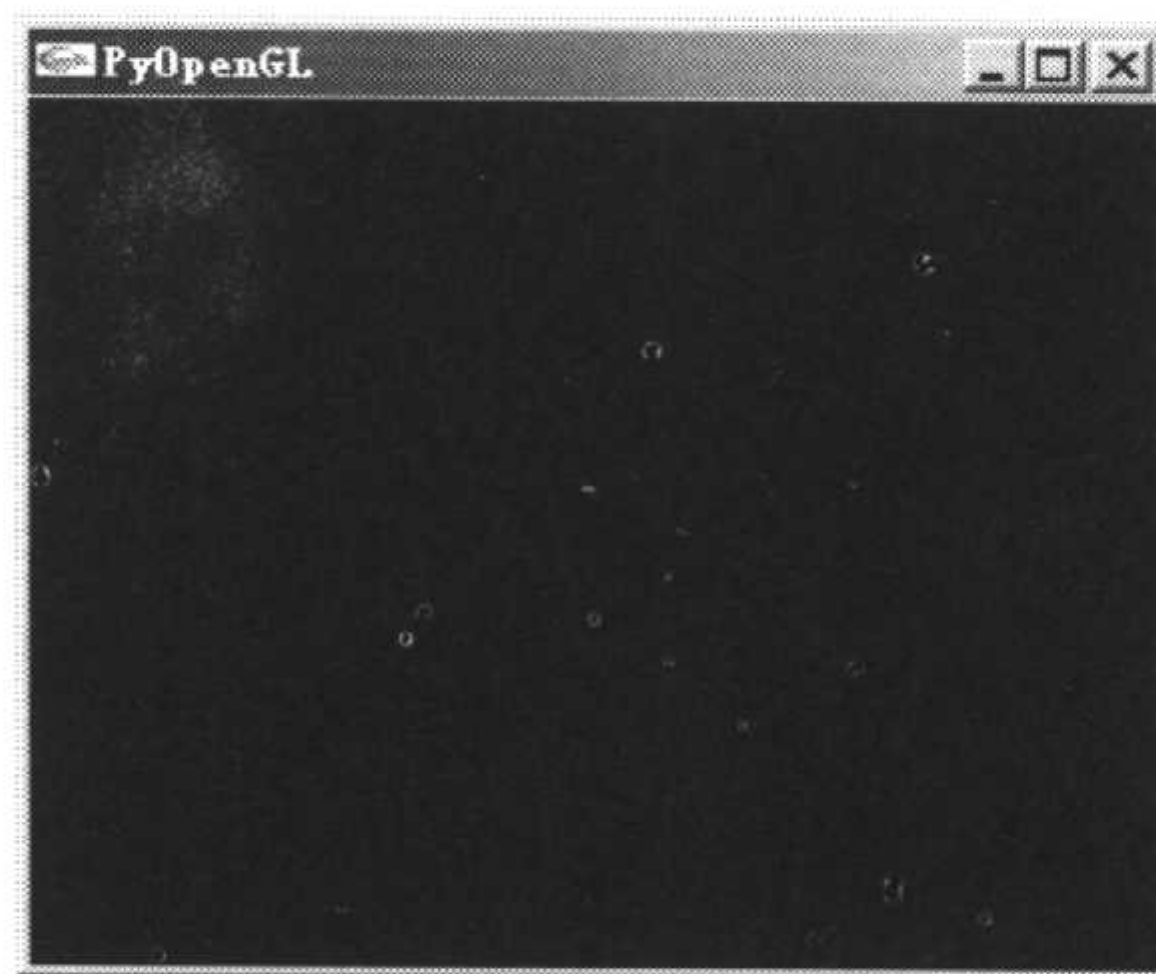


图 22-1 PyOpenGL 窗口

22.1.3 绘制文字

使用 PyOpenGL 绘制文字相对比较复杂, 需要使用 glutBitmapCharacter 函数, 其原型如下所示。

```
glutBitmapCharacter(font, character)
```

其参数含义如下。

- font: 所使用的字体。
- character: 输出字符的 ASCII 值。

可以选用的字体有如下几种。

- GLUT_BITMAP_8_BY_13
- GLUT_BITMAP_9_BY_15
- GLUT_BITMAP_TIMES_ROMAN_10
- GLUT_BITMAP_TIMES_ROMAN_24
- GLUT_BITMAP_HELVETICA_10
- GLUT_BITMAP_HELVETICA_12
- GLUT_BITMAP_HELVETICA_18

由于 glutBitmapCharacter 每次只能输出一个字符，因此需要在脚本中编写函数处理字符串。如下所示的 pyOpenGLText.py 使用 glutBitmapCharacter 在窗口中绘制文字。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLText.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:
    def _init_(self, width = 640, height = 480, title = 'PyOpenGL'):
        # 初始化
        # 传递命令行参数
        glutInit(sys.argv)
        # 设置显示模式
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
        # 设置窗口大小
        glutInitWindowSize(width, height)
        # 创建窗口
        self.window = glutCreateWindow(title)
        # 设置场景绘制函数
        glutDisplayFunc(self.Draw)
        # 调用 OpenGL 初始化函数
        self.InitGL(width, height)
        # 绘制场景
    def Draw(self):
        # 清除屏幕和深度缓存
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        # 重置观察矩阵
        glLoadIdentity()
        # 移动位置
        glTranslatef(0.0, 0.0, -1.0)
        # 设置颜色为绿色
        glColor3f(0.0, 1.0, 0.0)
        # 定位文字
        glRasterPos2f(0.0, 0.0)
        # 绘制文字
        self.DrawText('PyOpenGL')
        # 交换缓存
        glutSwapBuffers()
        # 绘制文字函数
    def DrawText(self, string):
        # 循环处理字符串
        for c in string:
            # 输出文字
            glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(c))
        # OpenGL 初始化函数
    def InitGL(self, width, height):
        # 设为黑色背景
        glClearColor(0.0, 0.0, 0.0, 0.0)
        # 设置深度缓存
        glClearDepth(1.0)
        # 设置深度测试类型
        glDepthFunc(GL_LESS)
        # 允许深度测试
        glEnable(GL_DEPTH_TEST)
        # 启动平滑阴影
        glShadeModel(GL_SMOOTH)
        # 设置观察矩阵
        glMatrixMode(GL_PROJECTION)
        # 重置观察矩阵
        glLoadIdentity()
```



```

        gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
        # 计算屏幕高宽比
        glMatrixMode(GL_MODELVIEW)
        # 设置观察矩阵
    def MainLoop(self):
        # 进入消息循环
        glutMainLoop()
    window = OpenGLWindow()
    # 创建窗口
    window.MainLoop()
    # 进入消息循环

```

运行 `glutBitmapCharacter` 脚本后如图 22-2 所示。

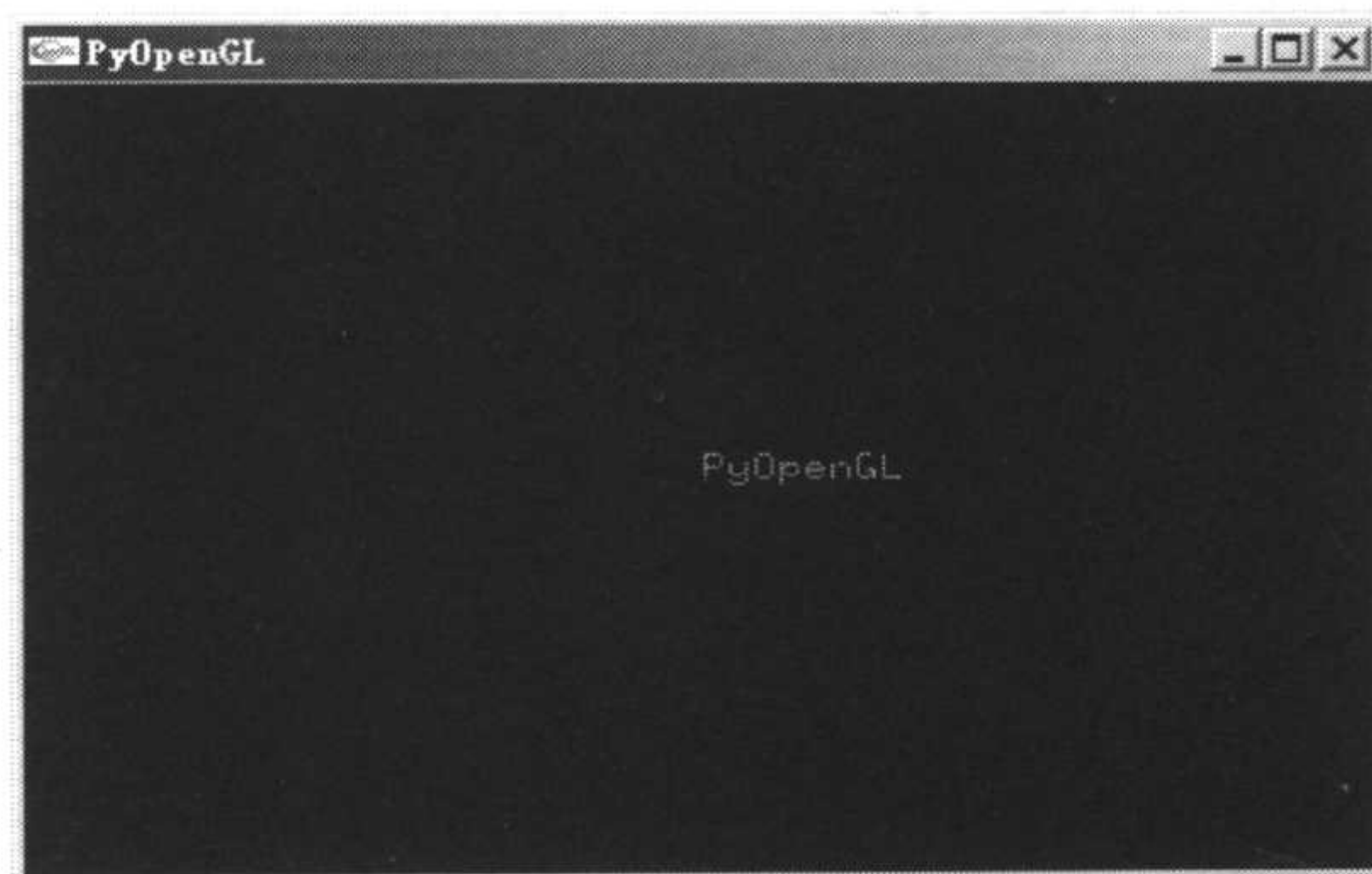


图 22-2 绘制文字

22.1.4 绘制 2D 图形

在 PyOpenGL 中绘制图形时应以 `glBegin` 函数开始，当绘制完成后应调用 `glEnd` 函数。
`glBegin` 函数原型如下所示。

```
glBegin (mode)
```

其参数含义如下。

- `mode`: 绘制的图形。

其中可以选择的图形有以下几种。

- `GL_POINTS`: 绘制点。
- `GL_LINES`: 绘制直线。
- `GL_LINE_STRIP`: 绘制连续直线，不封闭。
- `GL_LINE_LOOP`: 绘制连续直线，封闭。
- `GL_TRIANGLES`: 绘制三角形。
- `GL_TRIANGLE_STRIP`: 绘制三角形串。
- `GL_TRIANGLE_FAN`: 绘制三角扇形。
- `GL_QUADS`: 绘制四边形。
- `GL_QUAD_STRIP`: 绘制四边形串。
- `GL_POLYGON`: 绘制多边形。

由于在 PyOpenGL 中没有提供直接绘制圆形的函数，因此可以使用多边形来模拟绘制圆形。如下所示的 pyOpenGLDraw2D.py 脚本绘制了几种简单的 2D 图形。

```
# -*- coding:utf-8 -*-
# file: pyOpenGLDraw2D.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
import math
class OpenGLWindow:
    def _init_(self, width = 640, height = 480, title = 'PyOpenGL'):
        # 初始化
        # 传递命令行参数
        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
        # 设置显示模式
        # 设置窗口大小
        glutInitWindowSize(width, height)
        # 创建窗口
        self.window = glutCreateWindow(title)
        # 设置场景绘制函数
        glutDisplayFunc(self.Draw)
        # 调用 OpenGL 初始化函数
        self.InitGL(width, height)
        # 绘制场景
    def Draw(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        # 重置观察矩阵
        glLoadIdentity()
        # 移动位置
        glTranslatef(-2.0, 2.0, -6.0)
        # 绘制直线
        glBegin(GL_LINES)
        # 直线第一点坐标
        glVertex3f(0.0, 0.0, 0.0)
        # 直线第二点坐标
        glVertex3f(2.0, 0.0, 0.0)
        # 结束绘制
        glEnd()
        # 移动位置
        glTranslatef(3.0, 0.0, 0.0)
        # 通过绘制多边形来模拟圆
        glBegin(GL_POLYGON)
        i = 0
        while( i <= 3.14 *2 ):
            x = 0.5 * math.cos(i)
            y = 0.5 * math.sin(i)
            glVertex3f(x, y, 0.0)
            i = i + 0.01
        glEnd()
        # 移动位置
        glTranslatef(-2, -3.0, 0.0)
        # 绘制三角形
        glBegin(GL_POLYGON)
        glVertex3f(0.0, 1.0, 0.0)
        glVertex3f(1.0, -1.0, 0.0)
        glVertex3f(-1.0, -1.0, 0.0)
        glEnd()
        # 移动位置
        glTranslatef(2.5, 0.0, 0.0)
        # 绘制四边形
        glBegin(GL_QUADS)
        glVertex3f(-1.0, 1.0, 0.0)
        glVertex3f(1.0, 1.0, 0.0)
        glVertex3f(1.0, -1.0, 0.0)
        glVertex3f(-1.0, -1.0, 0.0)
```



```

    glEnd()
    glutSwapBuffers()
def InitGL(self, width, height):
    glClearColor(0.0, 0.0, 0.0, 0.0)
    glClearDepth(1.0)
    glDepthFunc(GL_LESS)
    glEnable(GL_DEPTH_TEST)
    glShadeModel(GL_SMOOTH)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)

    glMatrixMode(GL_MODELVIEW)
def MainLoop(self):
    glutMainLoop()
window = OpenGLWindow()
window.MainLoop()

```

交换缓存
OpenGL 初始化函数
设为黑色背景
设置深度缓存
设置深度测试类型
允许深度测试
启动平滑阴影
设置观察矩阵
重置观察矩阵
计算屏幕高宽比
设置观察矩阵
进入消息循环

创建窗口
进入消息循环

运行 pyOpenGLDraw2D.py 脚本后将创建如图 22-3 所示的窗口。

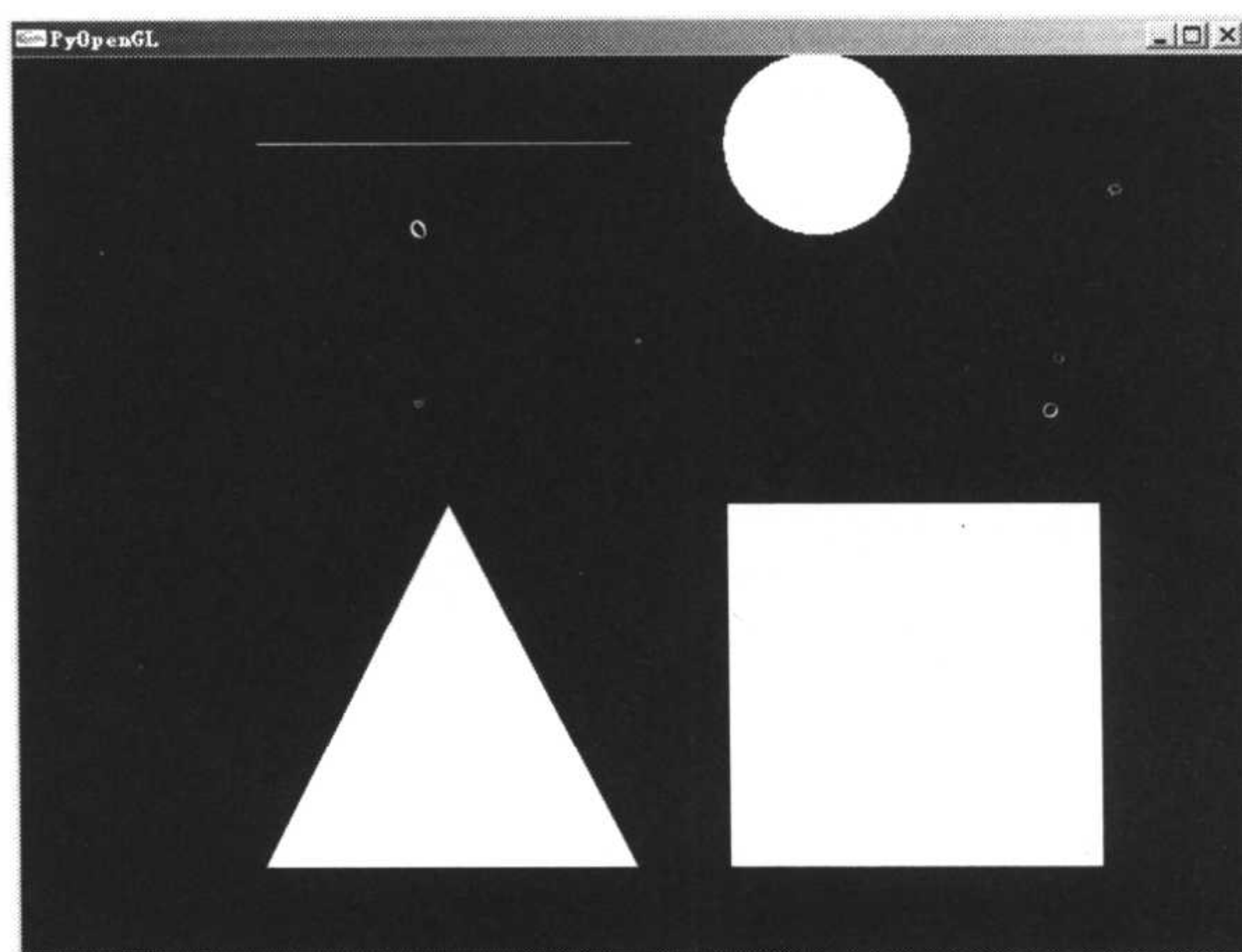


图 22-3 绘制 2D 图形

22.1.5 绘制 3D 图形

在 PyOpenGL 中绘制 3D 图形时和绘制 2D 图形类似，通过设置 Z 坐标，可以设置图形所在的空间位置。如下所示 pyOpenGLDraw3D.py 绘制了一个立方体。

```

# -*- coding:utf-8 -*-
# file: pyOpenGLDraw3D.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
class OpenGLWindow:

```

```

def _init_(self, width = 640, height = 480, title = 'PyOpenGL'):
    # 初始化
    glutInit(sys.argv)          # 传递命令行参数
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
    # 设置显示模式
    glutInitWindowSize(width, height)  # 设置窗口大小
    self.window = glutCreateWindow(title)  # 创建窗口
    glutDisplayFunc(self.Draw)         # 设置场景绘制函数
    self.InitGL(width, height)         # 调用 OpenGL 初始化函数

def Draw(self):
    # 绘制场景
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)  # 清除屏幕
    glLoadIdentity()  # 重置观察矩阵
    glTranslatef(1.5, 0.0, -7.0)  # 移动位置
    glRotatef(45, 1.0, 1.0, 1.0)  # 绕 X、Y、Z 轴旋转 45°
    glBegin(GL_QUADS)  # 开始绘制立方体
    glColor3f(1.0, 0.0, 0.0)  # 设置颜色为红色
    glVertex3f( 1.0, 1.0, -1.0)
    glVertex3f(-1.0, 1.0, -1.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f( 1.0, 1.0, 1.0)
    glColor3f(0.0, 1.0, 0.0)  # 设置颜色为绿色
    glVertex3f( 1.0, -1.0, 1.0)
    glVertex3f(-1.0, -1.0, 1.0)
    glVertex3f(-1.0, -1.0, -1.0)
    glVertex3f( 1.0, -1.0, -1.0)
    glColor3f(0.0, 0.0, 1.0)  # 设置颜色为蓝色
    glVertex3f( 1.0, 1.0, 1.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f(-1.0, -1.0, 1.0)
    glVertex3f( 1.0, -1.0, 1.0)
    glColor3f(1.0, 0.0, 0.0)
    glVertex3f( 1.0, -1.0, -1.0)
    glVertex3f(-1.0, -1.0, -1.0)
    glVertex3f(-1.0, 1.0, -1.0)
    glVertex3f( 1.0, 1.0, -1.0)
    glColor3f(0.0, 1.0, 0.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glVertex3f(-1.0, 1.0, -1.0)
    glVertex3f(-1.0, -1.0, -1.0)
    glVertex3f(-1.0, -1.0, 1.0)
    glColor3f(0.0, 0.0, 1.0)
    glVertex3f( 1.0, 1.0, -1.0)
    glVertex3f( 1.0, 1.0, 1.0)
    glVertex3f( 1.0, -1.0, 1.0)
    glVertex3f( 1.0, -1.0, -1.0)
    glEnd()
    glutSwapBuffers()  # 交换缓存

def InitGL(self, width, height):
    # OpenGL 初始化函数
    # 设为黑色背景
    glClearColor(0.0, 0.0, 0.0, 0.0)

```



```

glClearDepth(1.0)
glDepthFunc(GL_LESS)
glEnable(GL_DEPTH_TEST)
glShadeModel(GL_SMOOTH)
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)

glMatrixMode(GL_MODELVIEW)
def MainLoop(self):
    glutMainLoop()
window = OpenGLWindow()
window.MainLoop()

```

设置深度缓存
设置深度测试类型
允许深度测试
启动平滑阴影
设置观察矩阵
重置观察矩阵
计算屏幕高宽比
设置观察矩阵
进入消息循环
创建窗口
进入消息循环

运行 pyOpenGLDraw3D.py 脚本后将创建如图 22-4 所示的窗口。

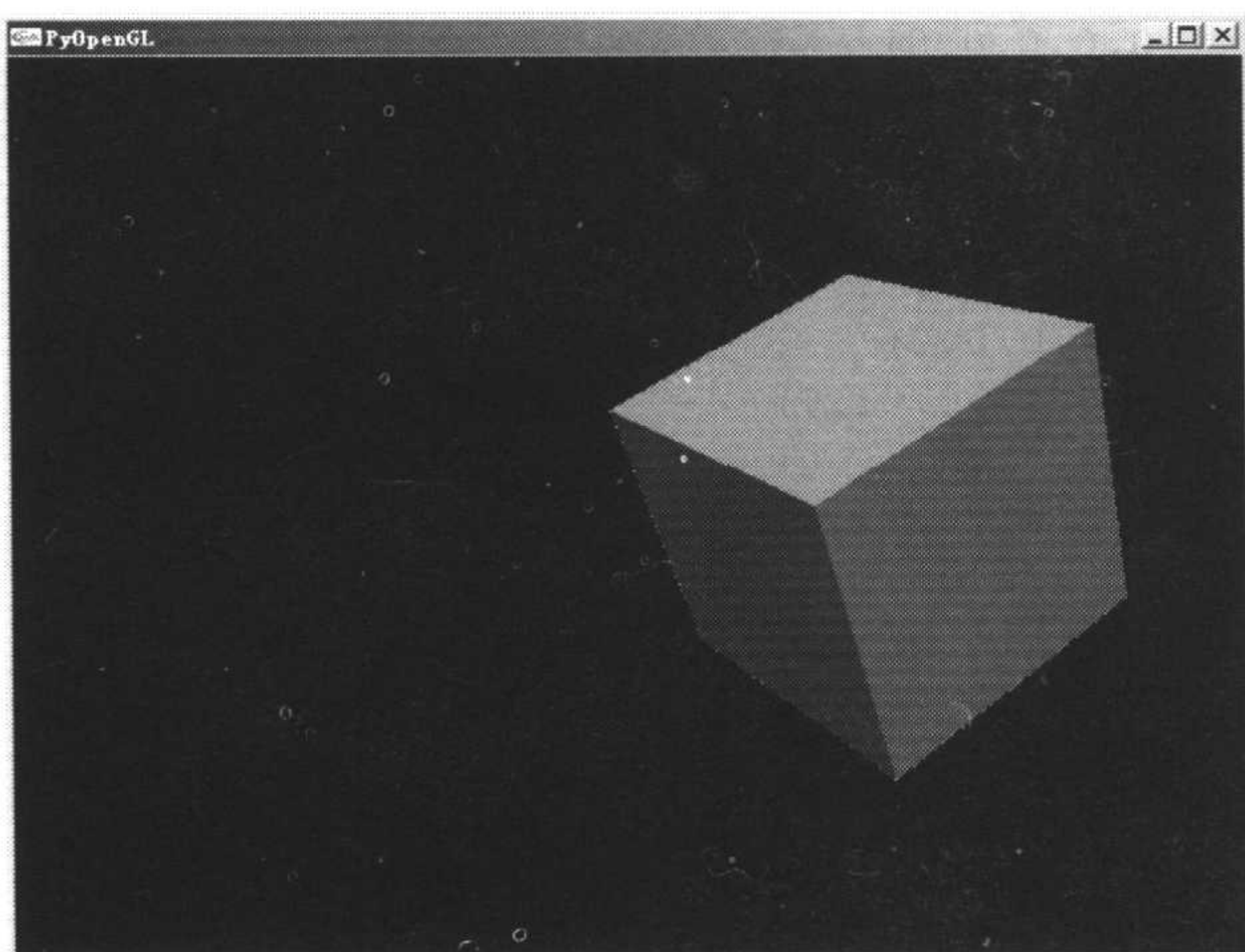


图 22-4 绘制 3D 图形

22.1.6 纹理映射

在 PyOpenGL 中进行纹理贴图需要使用 PIL 模块，在下一章中将讲解 PIL 模块详细的使用方法。如下所示的 pyOpenGLTexture.py 绘制了一个立方体，并对每一个面进行贴图。在 pyOpenGLTexture.py 脚本中使用 glutIdleFunc 函数，设置了空闲时的场景绘制函数，创建了立方体旋转的动画。

```

# -*- coding:utf-8 -*-
# file: pyOpenGLTexture.py
#
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import sys
import Image

```

```

class OpenGLWindow:
    def _init_(self, width = 640, height = 480, title = 'PyOpenGL'):
        # 初始化
        # 传递命令行参数
        glutInit(sys.argv)
        # 设置显示模式
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
        # 设置窗口大小
        glutInitWindowSize(width, height)
        # 创建窗口
        self.window = glutCreateWindow(title)
        # 设置场景绘制函数
        glutDisplayFunc(self.Draw)
        # 设置空闲时场景绘制函数
        glutIdleFunc(self.Draw)
        # 调用 OpenGL 初始化函数
        self.InitGL(width, height)
        # 旋转角度增量
        self.x = 0.2
        self.y = 0.2
        self.z = 0.2

    def Draw(self):
        # 绘制场景
        # 清除屏幕
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        # 重置观察矩阵
        glLoadIdentity()
        # 移动位置
        glTranslatef(0.0, 0.0, -5.0)
        # 绕 X 轴旋转
        glRotatef(self.x, 1.0, 0.0, 0.0)
        # 绕 Y 轴旋转
        glRotatef(self.y, 0.0, 1.0, 0.0)
        # 绕 Z 轴旋转
        glRotatef(self.z, 0.0, 0.0, 1.0)
        # 绘制立方体
        glBegin(GL_QUADS)
        # 对前面进行贴图
        glTexCoord2f(0.0, 0.0)
        glVertex3f(-1.0, -1.0, 1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f(1.0, -1.0, 1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f(1.0, 1.0, 1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f(-1.0, 1.0, 1.0)
        # 对后面进行贴图
        glTexCoord2f(1.0, 0.0)
        glVertex3f(-1.0, -1.0, -1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f(-1.0, 1.0, -1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f(1.0, 1.0, -1.0)
        glTexCoord2f(0.0, 0.0)
        glVertex3f(1.0, -1.0, -1.0)
        # 对顶面进行贴图
        glTexCoord2f(0.0, 1.0)
        glVertex3f(-1.0, 1.0, -1.0)
        glTexCoord2f(0.0, 0.0)
        glVertex3f(-1.0, 1.0, 1.0)
        glTexCoord2f(1.0, 0.0)
        glVertex3f(1.0, 1.0, 1.0)
        glTexCoord2f(1.0, 1.0)
        glVertex3f(1.0, 1.0, -1.0)
        # 对底面进行贴图
        glTexCoord2f(1.0, 1.0)
        glVertex3f(-1.0, -1.0, -1.0)
        glTexCoord2f(0.0, 1.0)
        glVertex3f(1.0, -1.0, -1.0)

```



```

    glTexCoord2f(0.0, 0.0)
    glVertex3f( 1.0, -1.0, 1.0)
    glTexCoord2f(1.0, 0.0)
    glVertex3f(-1.0, -1.0, 1.0)
    glTexCoord2f(1.0, 0.0) # 对右侧面进行贴图
    glVertex3f( 1.0, -1.0, -1.0)
    glTexCoord2f(1.0, 1.0)
    glVertex3f( 1.0, 1.0, -1.0)
    glTexCoord2f(0.0, 1.0)
    glVertex3f( 1.0, 1.0, 1.0)
    glTexCoord2f(0.0, 0.0)
    glVertex3f( 1.0, -1.0, 1.0)
    glTexCoord2f(0.0, 0.0) # 对左侧面进行贴图
    glVertex3f(-1.0, -1.0, -1.0)
    glTexCoord2f(1.0, 0.0)
    glVertex3f(-1.0, -1.0, 1.0)
    glTexCoord2f(1.0, 1.0)
    glVertex3f(-1.0, 1.0, 1.0)
    glTexCoord2f(0.0, 1.0)
    glVertex3f(-1.0, 1.0, -1.0)
    glEnd()
    glutSwapBuffers() # 交换缓存
    self.x = self.x + 0.2 # 旋转角度增加
    self.y = self.y + 0.2
    self.z = self.z + 0.2

def InitGL(self, width, height): # OpenGL 初始化函数
    self.LoadTextures() # 载入纹理
    glEnable(GL_TEXTURE_2D) # 允许纹理映射
    glClearColor(0.0, 0.0, 0.0, 0.0) # 设为黑色背景
    glClearDepth(1.0) # 设置深度缓存
    glDepthFunc(GL_LESS) # 设置深度测试类型
    glEnable(GL_DEPTH_TEST) # 允许深度测试
    glShadeModel(GL_SMOOTH) # 启动平滑阴影
    glMatrixMode(GL_PROJECTION) # 设置观察矩阵
    glLoadIdentity() # 重置观察矩阵
    gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
    # 计算屏幕高宽比
    glMatrixMode(GL_MODELVIEW) # 设置观察矩阵
def LoadTextures(self): # 载入纹理图片
    image = Image.open('python.bmp') # 打开图片
    width = image.size[0] # 图像宽度
    height = image.size[1] # 图像高度
    image = image.tostring('raw', 'RGBX', 0, -1) # 转换图像
    glBindTexture(GL_TEXTURE_2D, glGenTextures(1)) # 创建纹理
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1)
    glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height,
                 0, GL_RGBA, GL_UNSIGNED_BYTE, image)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)

```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)
def MainLoop(self):                                     # 进入消息循环
    glutMainLoop()
window = OpenGLWindow()                                  # 创建窗口
window.MainLoop()                                       # 进入消息循环

```

运行 pyOpenGLTexture.py 脚本后将创建如图 22-5 所示的窗口。

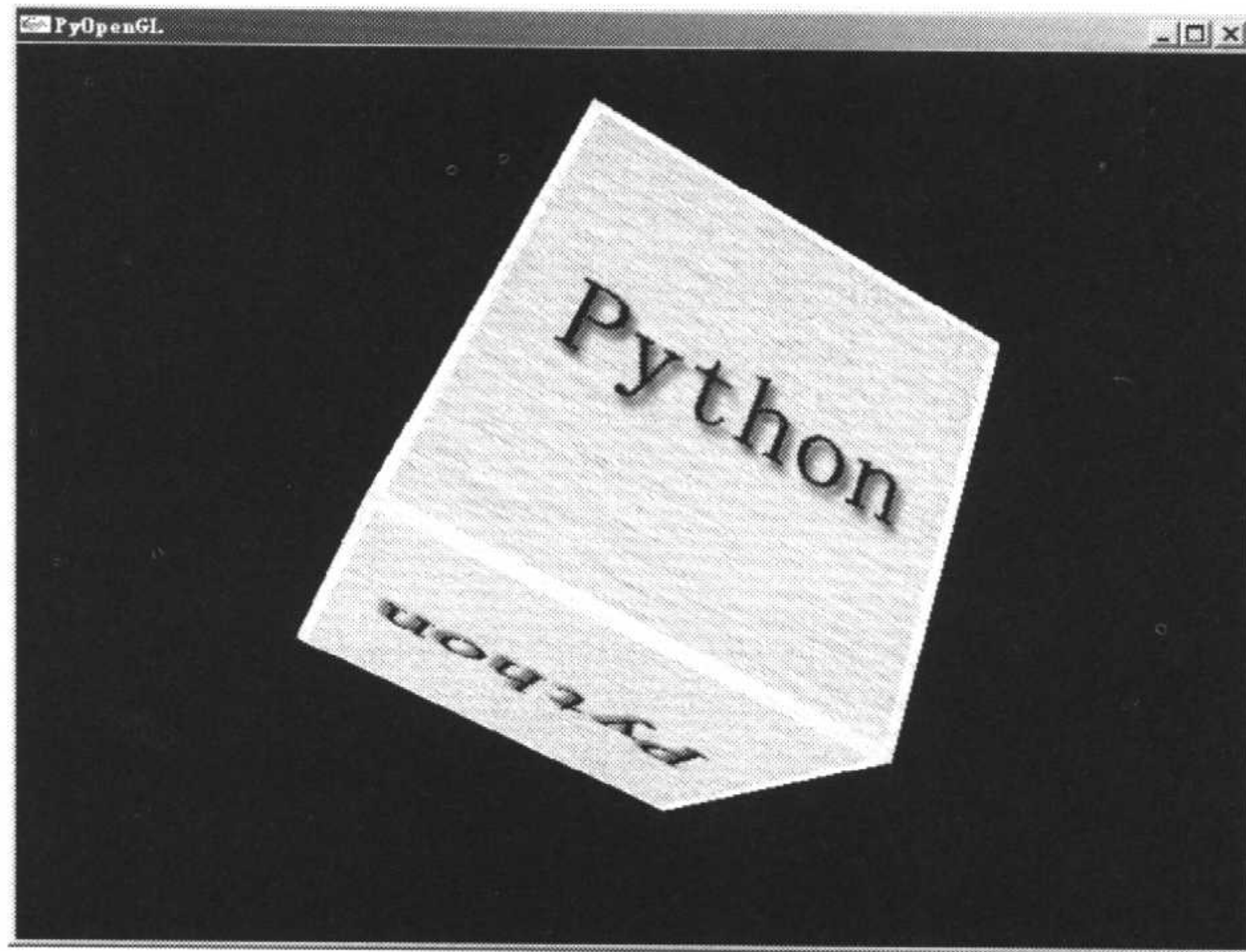


图 22-5 纹理贴图

22.2 播放音频文件

在 Windows 下，可以使用 PythonWin 提供的 win32com 模块，来调用 DirectSound 播放和处理音频文件。除此以外，还可以直接调用 WMPPlayer.OCX 来播放音频文件。

22.2.1 使用 DirectSound

DirectSound 是 DirectX API 的音频 (waveaudio) 组件之一，其提供了快速的混音、硬件加速功能，并且可以直接访问相关设备。DirectSound 允许进行波形声音的捕获、重放，也可以通过控制硬件和相应的驱动来获得更多的服务。在 Python 中，可以通过 PythonWin 中的 win32com 模块来使用 DirectSound。由于使用 DirectSound 需要处理大量的数据，Python 在运行速度上较慢。因此，此处仅给出一个简单的使用 DirectSound 播放 WAV 文件的例子。如下所示的 pyDirectSound.py 使用 DirectSound 播放 WAV 文件。

```

# -*- coding:utf-8 -*-
# file: pyDirectSound.py

```



```

#
import pywintypes
import struct
from win32com.directsound import directsound
import Tkinter
import tkFileDialog
WAV_HEADER_SIZE = struct.calcsize('<4sl4s4slhhllhh4sl')
class Window:
    def _init_(self):
        self.root = root = Tkinter.Tk()
        buttonAdd = Tkinter.Button(root, text = 'Add',
            command = self.add)
        buttonAdd.pack(side = 'left')
        buttonPlay = Tkinter.Button(root, text = 'Play',
            command = self.play)
        buttonPlay.pack(side = 'left')
        buttonStop = Tkinter.Button(root, text = 'Stop',
            command = self.stop)
        buttonStop.pack(side = 'left')
    def MainLoop(self):
        self.root.mainloop()
    def add(self):
        self.file = tkFileDialog.askopenfilename(
            title = 'Python DirectSound',
            filetypes=[('WAV', '*.wav')])
    def play(self):
        f = open(self.file, 'rb')
        header = f.read(WAV_HEADER_SIZE)
        (riff, riffsize, wave, fmt, fmtsize,
            format, nchannels, samplespersecond,
            datarate, blockalign, bitspersample,
            data, size) = \
            struct.unpack('<4sl4s4slhhllhh4sl', header)
        if riff != 'RIFF' or fmtsize != 16 or fmt != 'fmt' or data != 'data':
            raise 'Data Error'
        wfx = pywintypes.WAVEFORMATEX()
        wfx.wFormatTag = format
        wfx.nChannels = nchannels
        wfx.nSamplesPerSec = samplespersecond
        wfx.nAvgBytesPerSec = datarate
        wfx.nBlockAlign = blockalign
        wfx.wBitsPerSample = bitspersample
        d = directsound.DirectSoundCreate(None, None)
        d.SetCooperativeLevel(None, directsound.DSSCL_PRIORITY)
        sdesc = directsound.DSBUFFERDESC()
        sdesc.dwFlags = (
            directsound.DSBCAPS_STICKYFOCUS |
            directsound.DSBCAPS_CTRLPOSITIONNOTIFY

```

导入模块

设置 WAV 头

创建组件

进入消息循环

添加播放文件

播放文件

打开文件

读取 WAV 文件头

获取参数值

判断文件格式

创建 WAVEFORMATEX 结构

用 DirectSound 播放声音

```

    )
    sdesc.dwBufferBytes = size
    sdesc.lpwfxFormat = wfx
    self.buffer = buffer = d.CreateSoundBuffer(sdesc, None)
    buffer.Update(0, f.read(size))
    buffer.Play(0)
    def stop(self):
        self.buffer.Stop() # 停止
window = Window()
window.MainLoop()

```

22.2.2 使用 WMPlayer.OCX

使用 DirectSound 播放音频文件比较麻烦,而且需要自己对文件进行解码。由于 PythonWin 提供了对 COM 组件的支持,因此可以在 Python 中直接使用 WMPlayer.OCX 组件来播放音频文件。如下所示的 pyMusicPlayer.py 使用 WMPlayer.OCX 组件创建了一个简单的音乐播放器。

```

# -*- coding:utf-8 -*-
# file: pyMusicPlayer.py
#
import Tkinter # 导入模块
import tkFileDialog
from win32com.client import Dispatch
class Window:
    def __init__(self):
        self.root = root = Tkinter.Tk() # 创建窗口
        buttonAdd = Tkinter.Button(root, text = 'Add',
            command = self.add)
        buttonAdd.place(x = 150, y = 15)
        buttonPlay = Tkinter.Button(root, text = 'Play',
            command = self.play)
        buttonPlay.place(x = 200, y = 15)
        buttonPause = Tkinter.Button(root, text = 'Pause',
            command = self.pause)
        buttonPause.place(x= 250, y = 15)
        buttonStop = Tkinter.Button(root, text = 'Stop',
            command = self.stop)
        buttonStop.place(x= 300, y = 15)
        buttonNext = Tkinter.Button(root, text = 'Next',
            command = self.next)
        buttonNext.place(x = 350, y = 15)
        frame = Tkinter.Frame(root, bd=2)
        self.playList = Tkinter.Text(frame)
        scrollbar = Tkinter.Scrollbar(frame)
        scrollbar.config(command=self.playList.yview)
        self.playList.pack(side = Tkinter.LEFT)
        scrollbar.pack(side=Tkinter.RIGHT, fill=Tkinter.Y)
        frame.place(y = 50)
        self.wmp = Dispatch('WMPlayer.OCX') # 绑定 WMPlayer.OCX

```



```
def MainLoop(self):
    self.root.minsize(510,380)
    self.root.maxsize(510,380)
    self.root.mainloop()
def add(self):
    file = tkFileDialog.askopenfilename(title = 'Python Music Player',
        filetypes=[('MP3', '*.mp3'),
            ('WMA', '*.wma'), ('WAV', '*.wav')])
    if file:
        media = self.wmp.newMedia(file)
        self.wmp.currentPlaylist.appendItem(media)
        self.playList.insert(Tkinter.END, file + '\n')
def play(self):
    self.wmp.controls.play()
def pause(self):
    self.wmp.controls.pause()
def next(self):
    self.wmp.controls.next()
def stop(self):
    self.wmp.controls.stop()
window = Window()
window.MainLoop()
```

进入消息循环

添加播放文件

播放文件

暂停

下一首

停止

运行 pyMusicPlayer.py 脚本如图 22-6 所示。单击【Add】按钮，可以向播放列表中添加文件，单击【Play】按钮，将播放列表中的音乐。

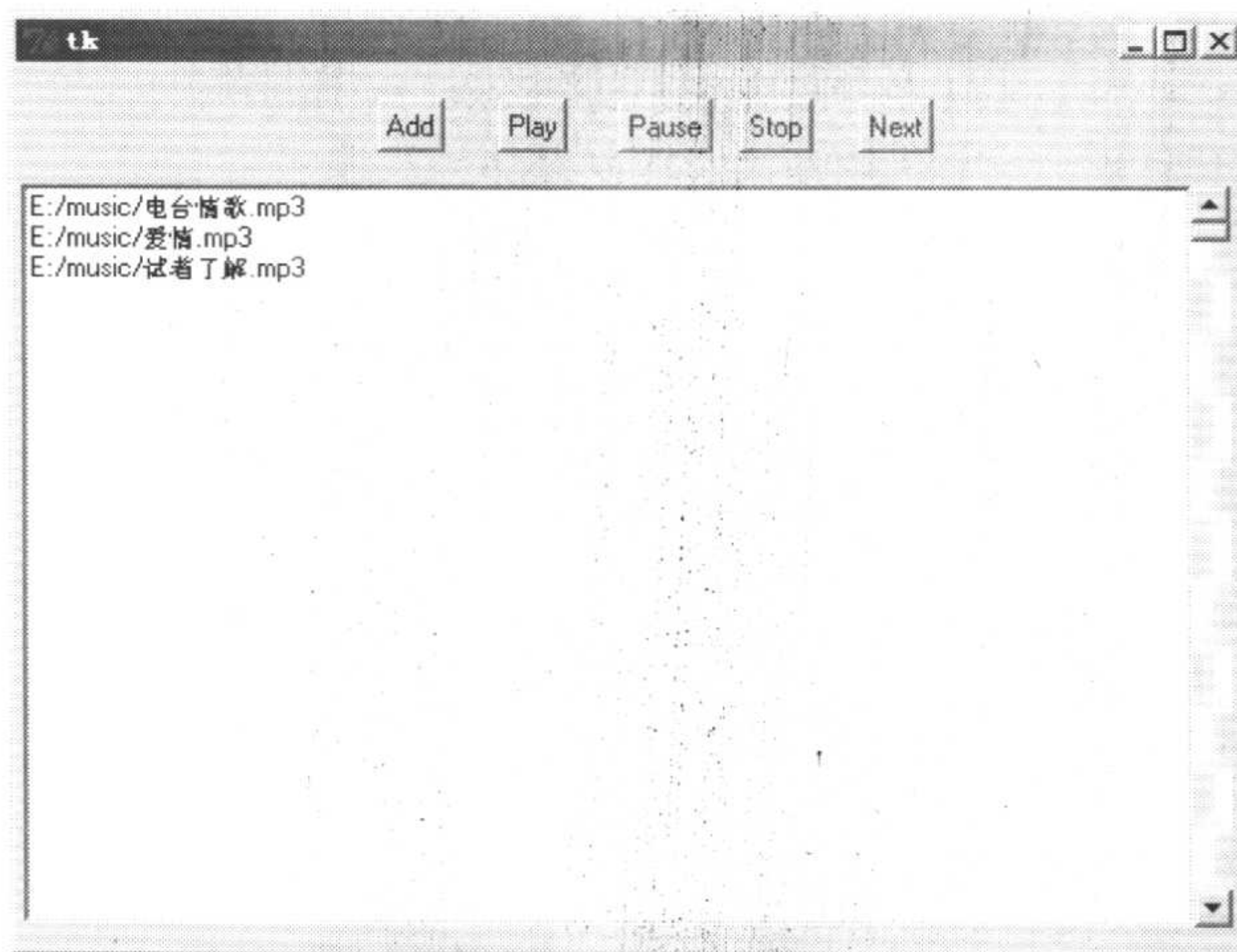


图 22-6 使用 WMPPlayer.OCX 播放音乐

22.3 PyGame

PyGame 是用来编写游戏的 Python 模块。PyGame 是基于 SDL 的，SDL (Simple DirectMedia Layer) 是一个跨平台的多媒体开发包，SDL 专门为游戏和多媒体设计。使用 PyGame 可以创

建使用 SDL 库创建的游戏和多媒体程序。

22.3.1 安装 PyGame

PyGame 是跨平台的 Python 模块，PyGame 的官方网站提供了 Windows 下的安装程序，以 Python 2.5 为例，其安装过程如下所示。

- (1) 从 PyGame 官方网站 <http://www.pygame.org> 下载 Windows 下的安装程序 pygame-1.7.1release.win32-py2.5.exe。
- (2) 双击运行安装程序，如图 22-7 所示。
- (3) 单击【下一步】按钮，进入安装路径选择界面，如图 22-8 所示。

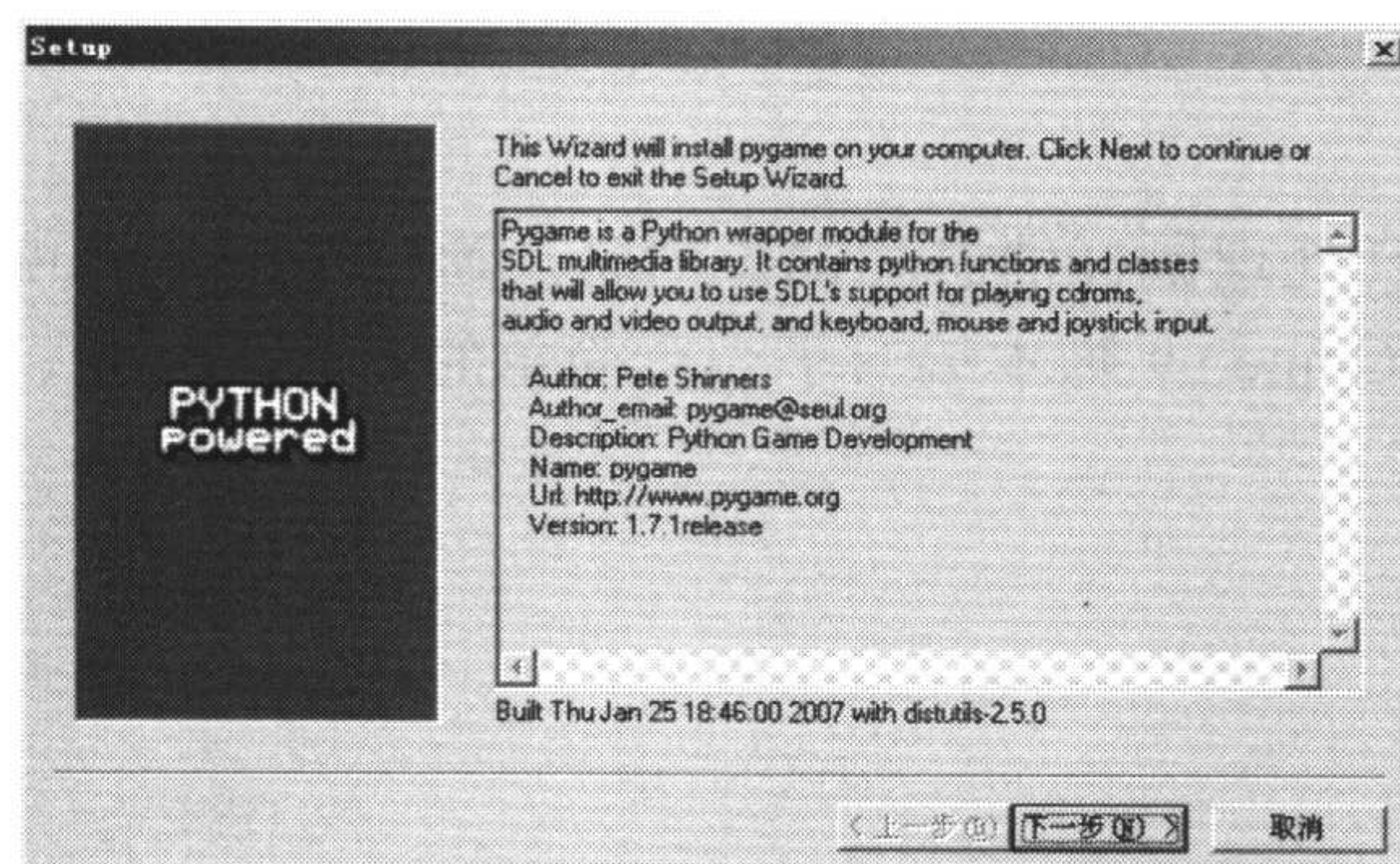


图 22-7 PyGame 安装程序

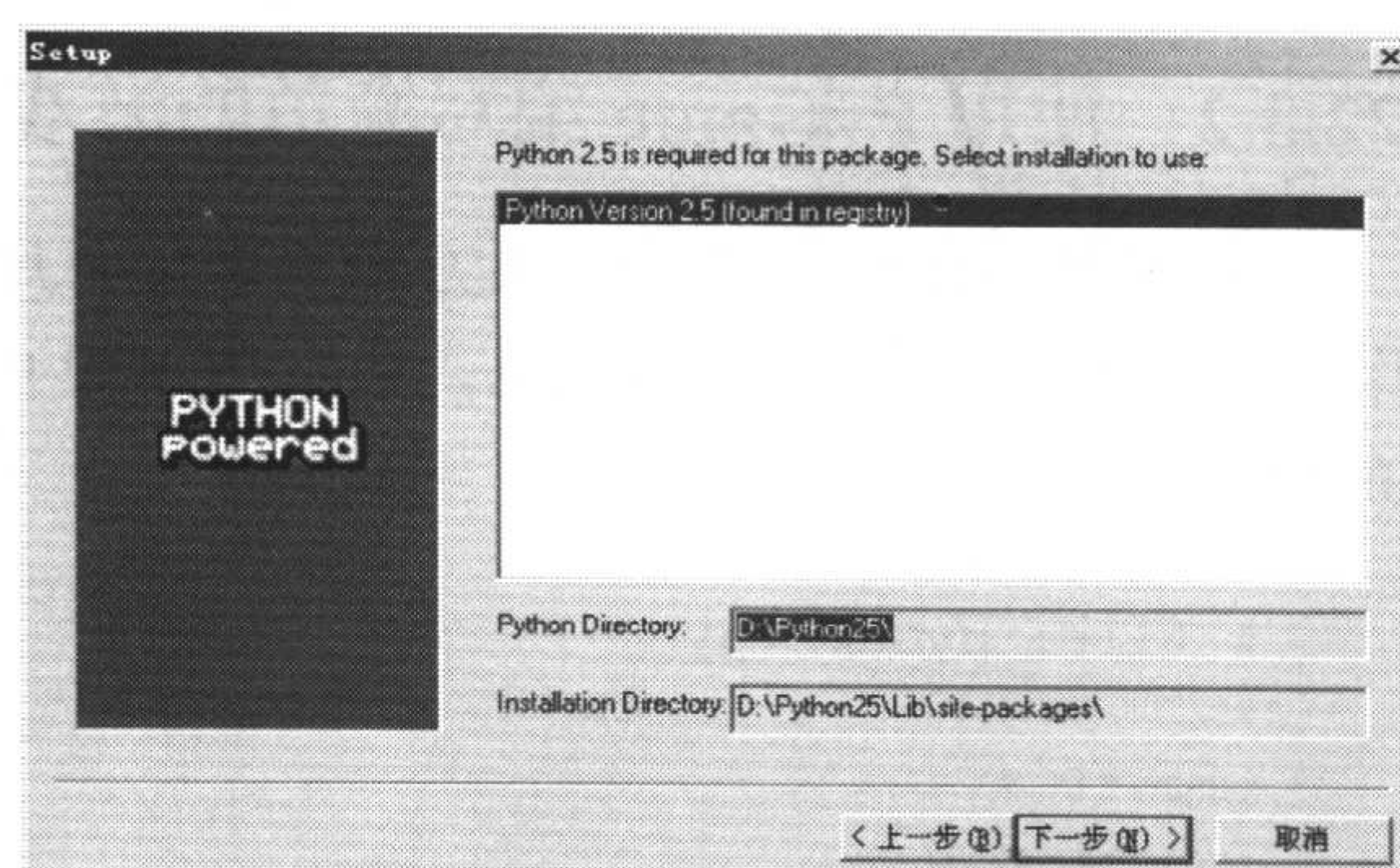


图 22-8 路径选择

- (4) 单击【下一步】按钮，进入安装确认界面，如图 22-9 所示。单击【下一步】按钮，完成 PyGame 的安装。

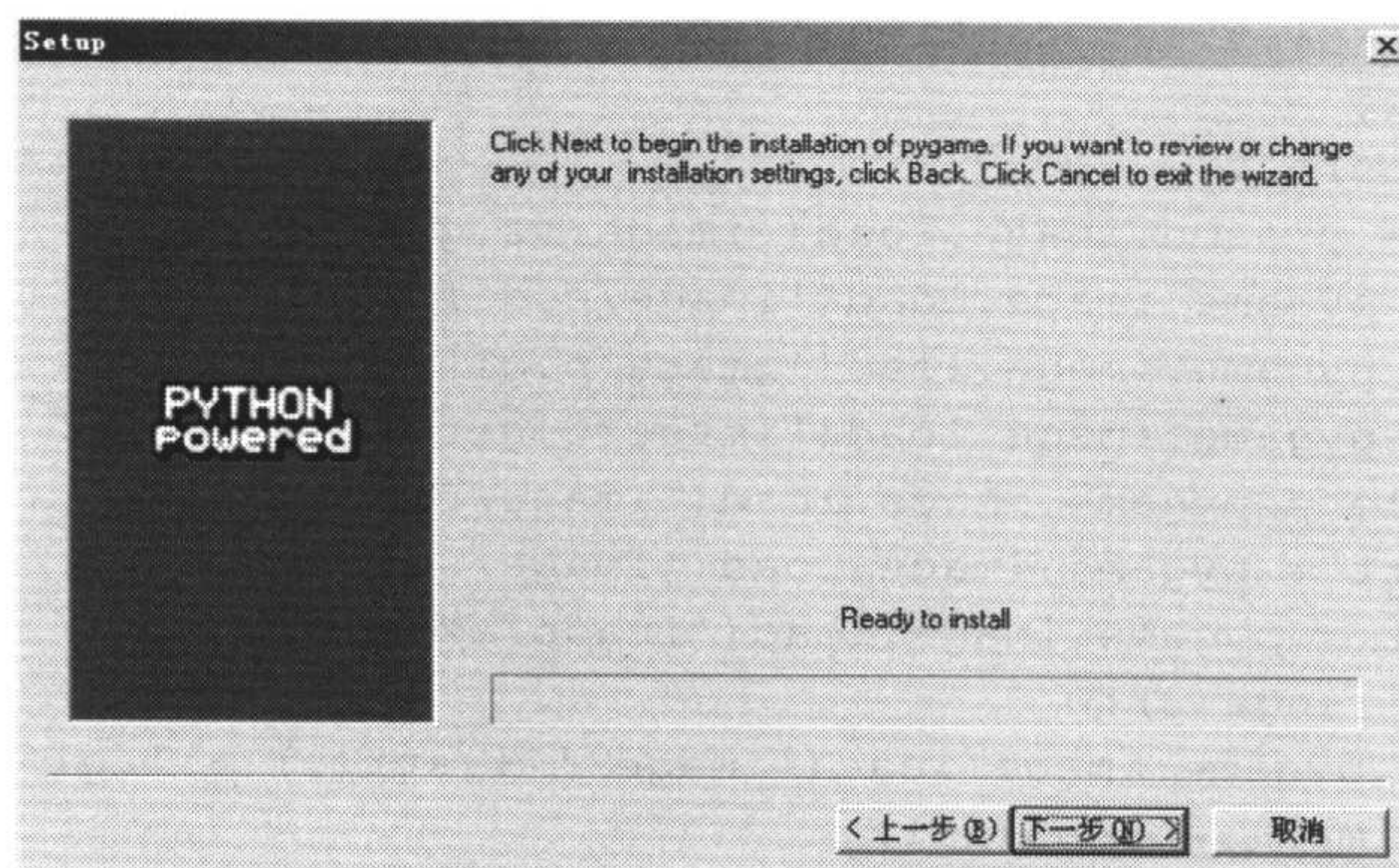


图 22-9 确认安装

在 PyGame 中将各种对游戏的支持分成更小的模块，常用的模块具体如下所示。

- Display: 提供了与屏幕显示相关的函数。
- Event: 提供了处理事件的函数。
- Image: 提供了处理图片的函数。
- Key: 提供了处理键盘按键的相关函数。
- Mouse: 提供了处理鼠标消息的相关函数。
- Movie: 提供了播放视频文件的相关函数, 需要 PyMedia 的支持。
- Music: 提供了播放音频文件的相关函数。
- Surface: 提供了绘制屏幕相关的函数。
- Time: 提供了处理时间的相关函数。

22.3.2 使用 PyGame 编写简单的游戏

使用 PyGame 模块可以很方便地绘制图片、处理鼠标消息和键盘消息。通过 PyGame 可以很快地编写出简单的小游戏。如下所示的 pyGame.py 脚本编写了一个简单的石头、剪子、布的游戏。

```
# -*- coding:utf-8 -*-
# file: pyGame.py
#
import sys
import pygame
import threading
import random

class Game:                                     # 创建游戏类
    def __init__(self):
        pygame.init()                          # pygame 初始化
        self.screen = pygame.display.set_mode((800,600)) # 设置显示模式
        pygame.display.set_caption('Python Game')      # 设置窗口标题
        self.image = []                        # 图片列表
        self.imagerect = []                   # 图片大小列表
        self.vs = pygame.image.load('image/vs.gif')   # 载入图片
        self.o = pygame.image.load('image/o.gif')
        self.p = pygame.image.load('image/p.gif')
        self.u = pygame.image.load('image/u.gif')
        self.title = pygame.image.load('image/title.gif')
        self.start = pygame.image.load('image/start.gif')
        self.exit = pygame.image.load('image/exit.gif')
        for i in range(3):
            gif = pygame.image.load('image/' + str(i) + '.gif')
            self.image.append(gif)
        for i in range(3):                     # 处理图片绘制区域
            image = self.image[i]
            rect = image.get_rect()
            rect.left = 200 * (i+1) + rect.left
```

```

        rect.top = 400
        self.imagerect.append(rect)
def Start(self):
    self.screen.blit(self.title, (200,100,400,140))
    self.screen.blit(self.start, (350,300,100,50))
    self.screen.blit(self.exit, (350, 400,100,50))
    pygame.display.flip()
    start = 1
    while start:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if self.isStart() == 0:
                    start = 0
                elif self.isStart() == 1:
                    sys.exit()
                else:
                    pass
            else:
                pass
        self.run()
def run(self):
    self.screen.fill((0,0,0))
    for i in range(3):
        self.screen.blit(self.image[i], self.imagerect[i])
    pygame.display.flip()
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                self.OnMouseButtonDown()
            else:
                pass
def isStart(self):
    pos = pygame.mouse.get_pos()
    if pos[0] > 350 and pos[0] < 450:
        if pos[1] > 300 and pos[1] < 350:
            return 0
        elif pos[1] > 400 and pos[1] < 450:
            return 1
        else:
            return 2
    else:
        return 2
def OnMouseButtonDown(self):
    self.screen.blit(self.vs, (300, 150, 140, 140))
    pos = pygame.mouse.get_pos()

```

绘制游戏初始界面
 # 绘制游戏名称
 # 绘制开始按钮
 # 绘制退出按钮
 # 刷新屏幕

 # 进入消息循环
 # 处理消息

 # 处理鼠标单击消息

 # 开始游戏
 # 开始游戏

 # 绘制图片
 # 刷新屏幕进入消息循环

 # 处理消息

 # 处理鼠标单击消息

 # 判断鼠标单击的按钮

 # 处理鼠标单击消息
 # 绘制图片
 # 获取鼠标位置


```

if pos[1] > 400 and pos[1] < 540:
    if pos[0] > 200 and pos[0] < 340:
        self.screen.blit(self.image[0],
                           (150, 150, 140, 140))
        self.isWin(0)
    elif pos[0] > 400 and pos[0] < 540:
        self.screen.blit(self.image[1],
                           (150, 150, 140, 140))
        self.isWin(1)
    elif pos[0] > 600 and pos[0] < 740:
        self.screen.blit(self.image[2],
                           (150, 150, 140, 140))
        self.isWin(2)
    else:
        pass
def isWin(self, value):
    num = random.randint(0, 2)
    self.screen.blit(self.image[num],
                      (450, 150, 590, 240))
    pygame.display.flip()
    if num == value:
        self.screen.blit(self.o,
                           (220, 10, 140, 70))
        pygame.display.flip()
    elif num < value:
        if num == 0:
            if value == 2:
                self.screen.blit(self.u,
                                   (220, 10, 140, 70))
            else:
                self.screen.blit(self.p,
                                   (220, 10, 140, 70))
            pygame.display.flip()
        else:
            self.screen.blit(self.u,
                               (220, 10, 140, 70))
            pygame.display.flip()
    else:
        if num == 2:
            if value == 1:
                self.screen.blit(self.u,
                                   (220, 10, 140, 70))
            else:
                self.screen.blit(self.p,
                                   (220, 10, 140, 70))
            pygame.display.flip()
        else:
            self.screen.blit(self.u,
                               (220, 10, 140, 70))

```

判断鼠标位置

判断谁赢
产生随机数
绘制相应图片

刷新屏幕
判断谁赢

```
pygame.display.flip()  
game = Game()  
game.Start()
```

运行 pyGame.py 脚本后, 如图 22-10 所示, 单击【开始】按钮, 如图 22-11 所示。游戏开始后分别单击相应的图片将出拳, 如图 22-12 所示。

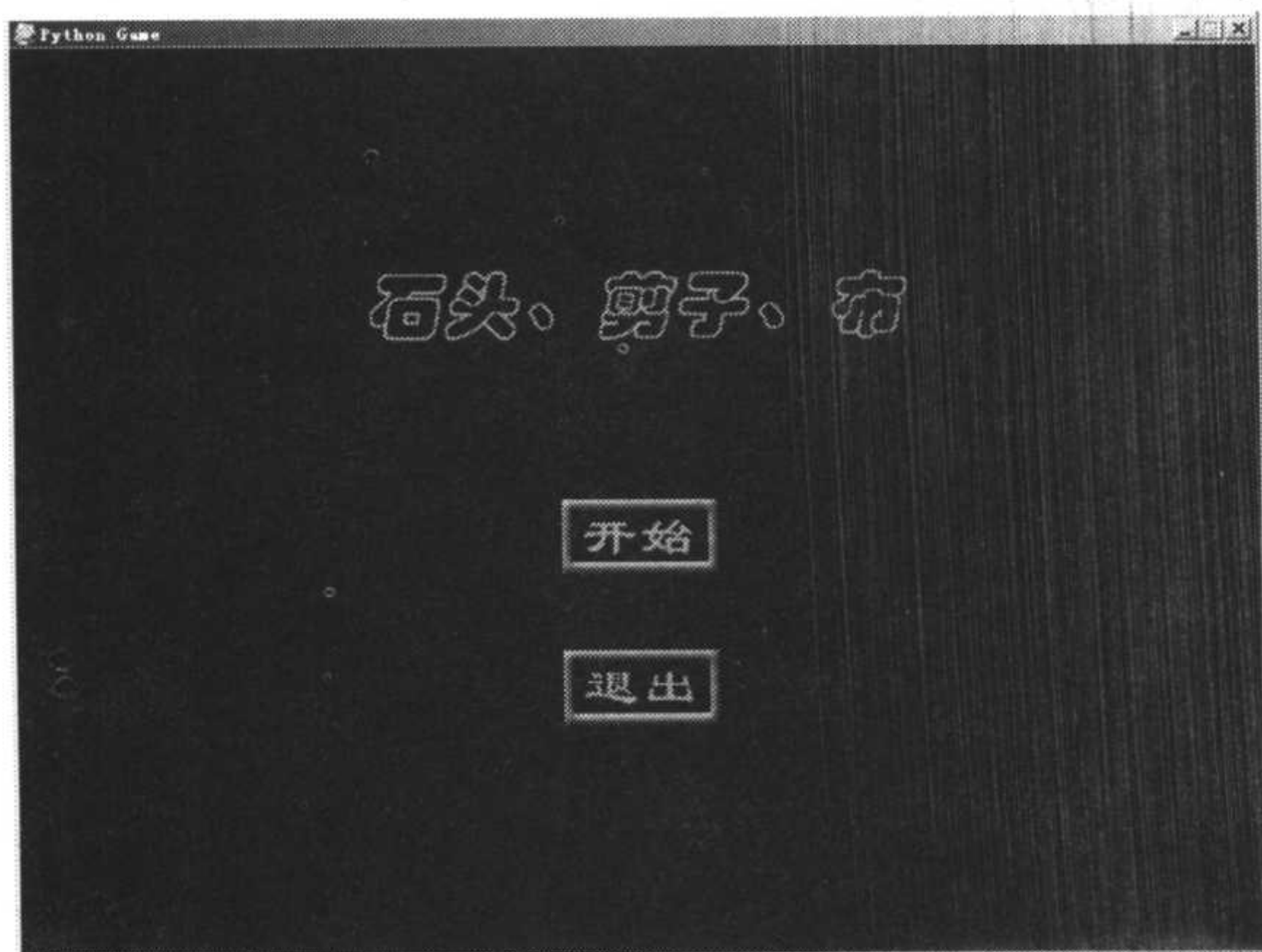


图 22-10 游戏开始界面

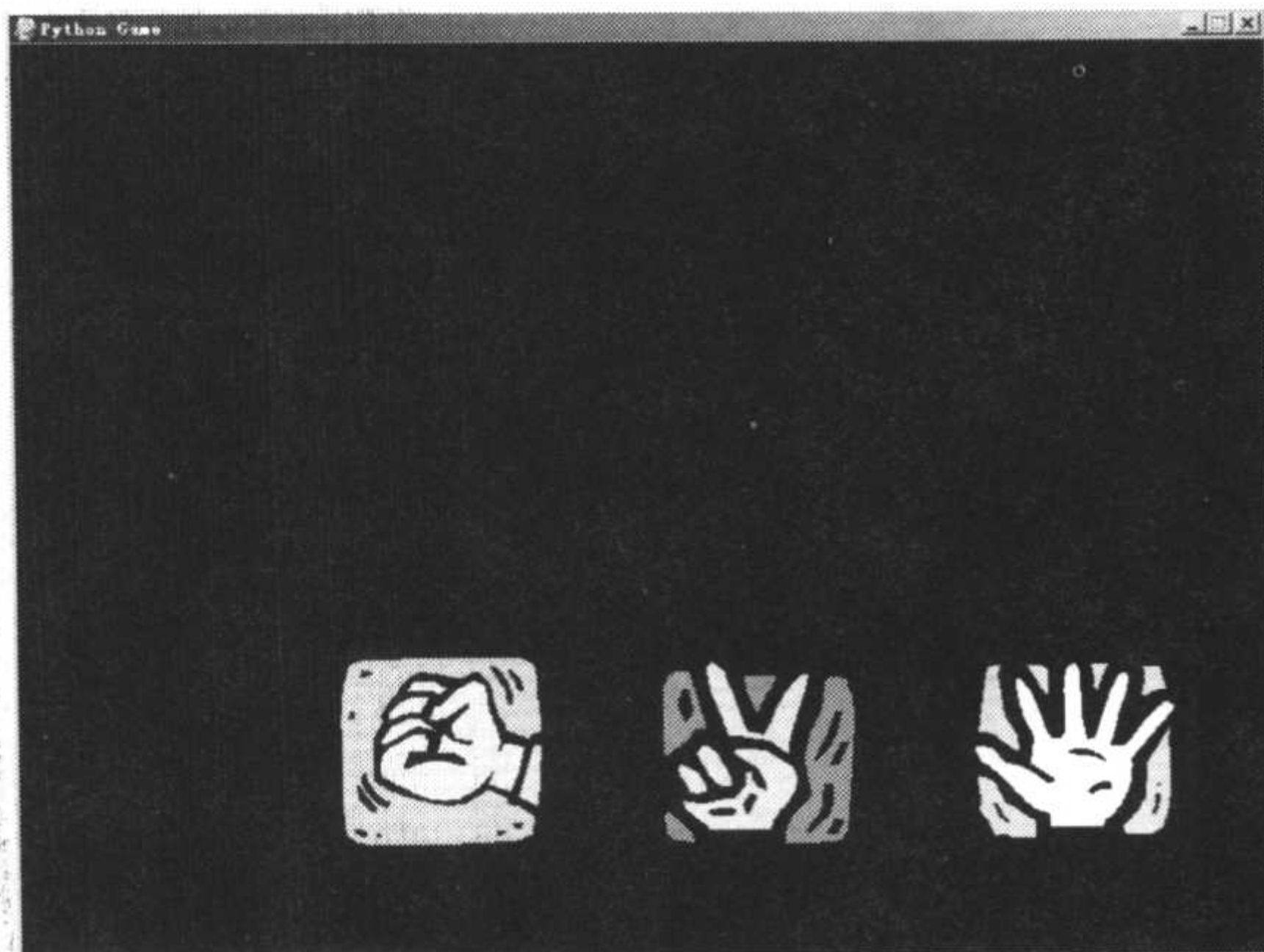


图 22-11 游戏界面

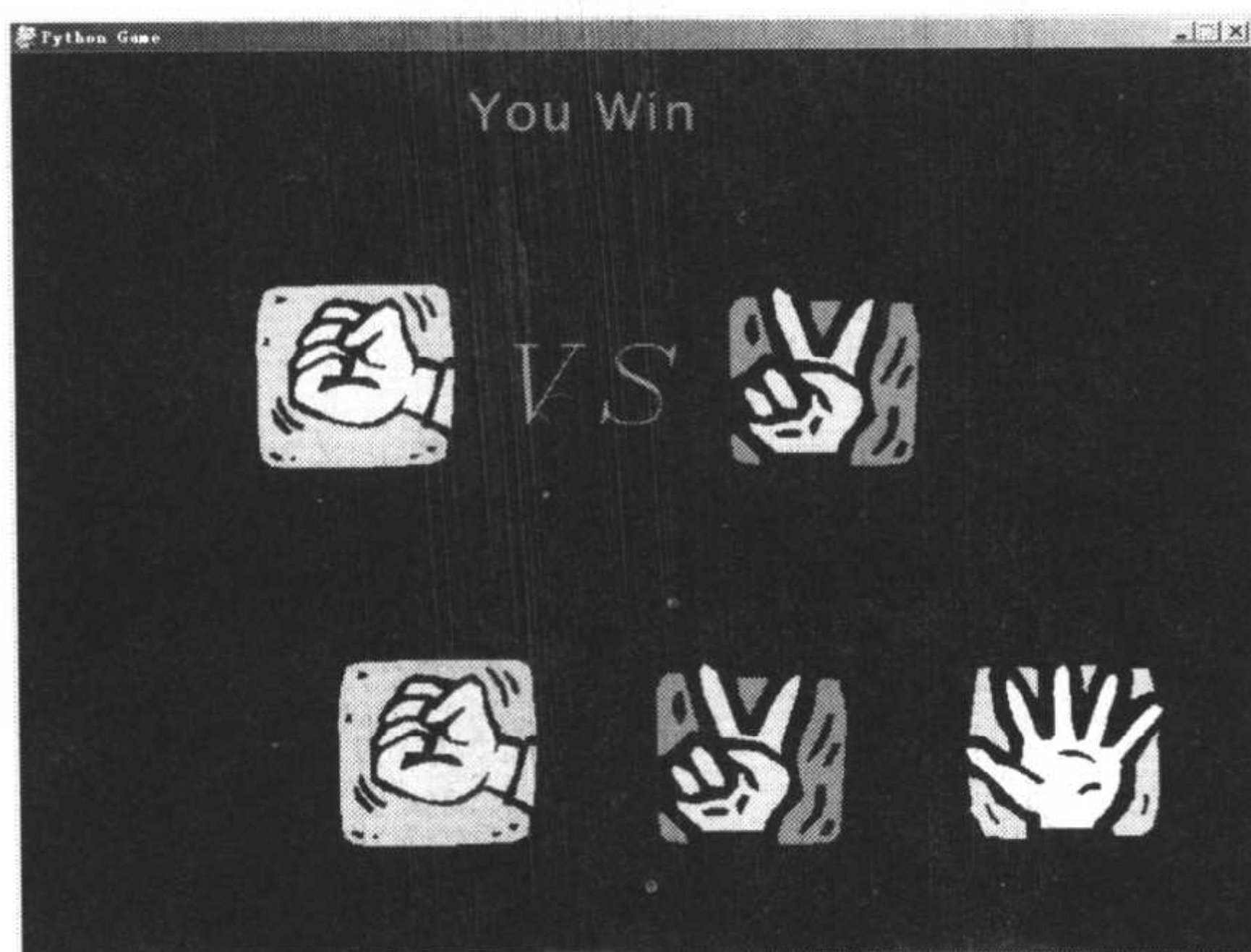


图 22-12 进行游戏

第 23 章 使用 PIL 处理图片

PIL (Python Imaging Library) 为 Python 提供了强大的图形处理的能力, 并支持多种图形文件格式。通过使用 PIL 模块, 可以使用 Python 对图片进行处理。

23.1 PIL 概述

PIL 是跨平台的, 在 Windows 下可以使用 PIL 的强大功能。由于 PIL 不是 Python 自带的模块, 因此需要用户自己安装。在 Windows 下安装 PIL 十分简便。

23.1.1 安装 PIL

PIL 官方网站提供了 Windows 下的安装程序, 以 Python 2.5 为例, 其安装过程如下所示。

- (1) 从 PIL 官方网站下载 Windows 下的安装程序 PIL-1.1.6.win32-py2.5.exe。
- (2) 双击运行安装程序, 如图 23-1 所示。
- (3) 单击【下一步】按钮, 进入安装路径选择界面, 如图 23-2 所示。

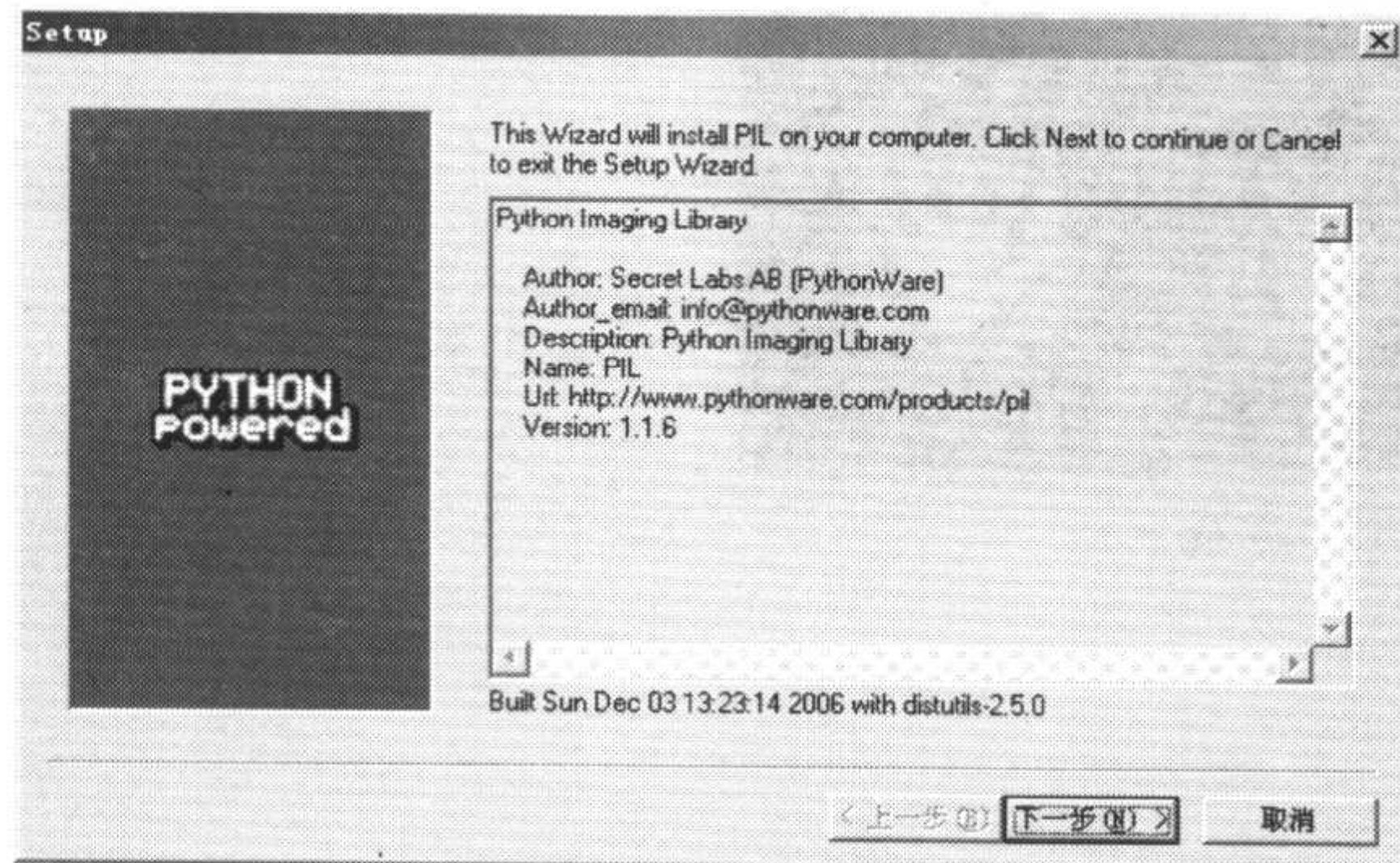


图 23-1 PIL 安装程序

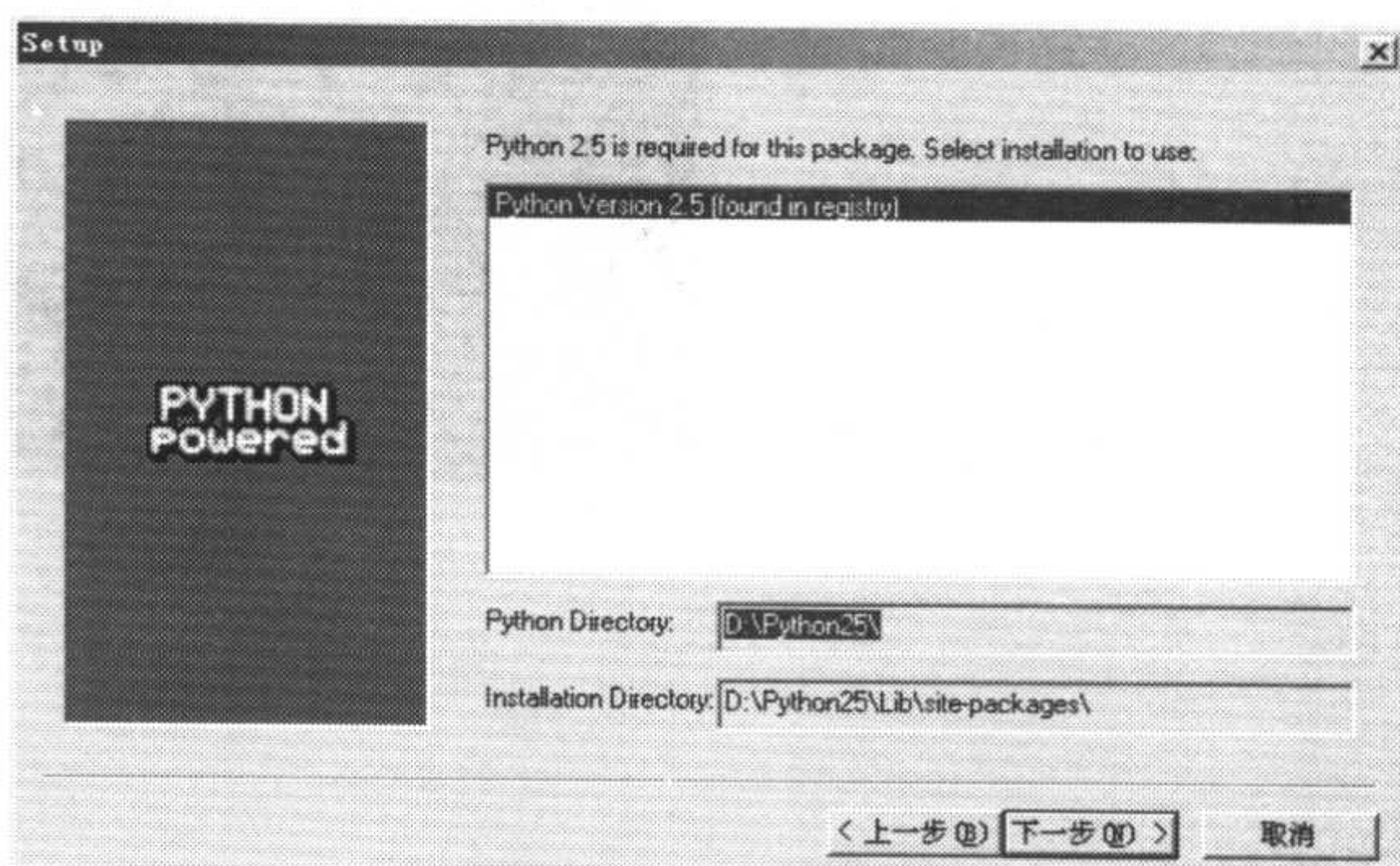


图 23-2 选择路径

(4) 单击【下一步】按钮进入安装确认界面, 如图 23-3 所示。单击【下一步】按钮, 完成 PIL 的安装。

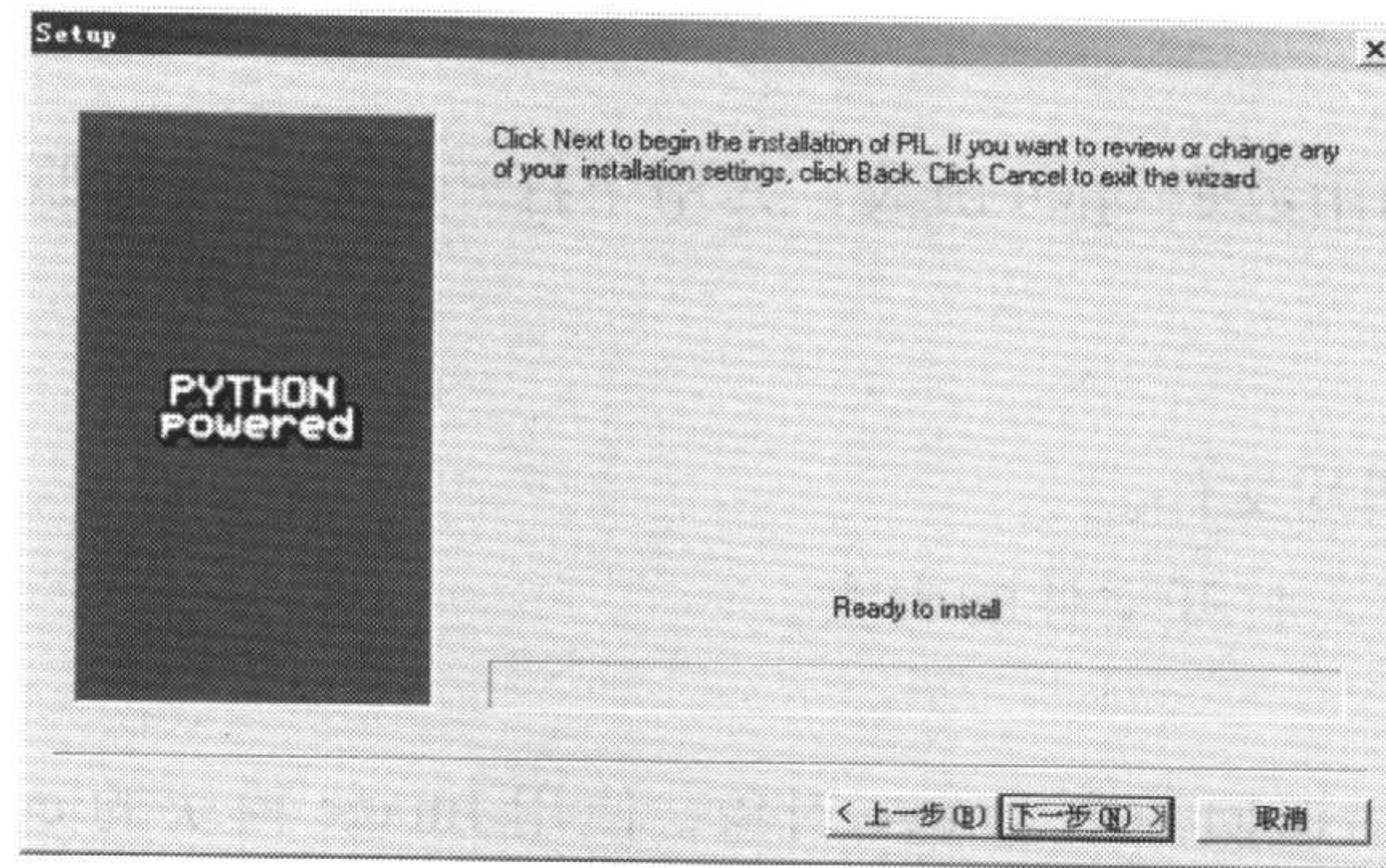


图 23-3 确认安装

23.1.2 PIL 简介

在 PIL 中提供了以下主要模块对图片进行处理。

- Image: PIL 的主要模块。
- ImageChops: 图片计算模块。
- ImageColor: 颜色模块。
- ImageDraw: 绘图模块。
- ImageEnhance: 图片效果模块。
- ImageFile: 图片文件存取模块。
- ImageFileIO: 图片流模块。
- ImageFilter: 图片过滤模块。
- ImageFont: 字形模块, 用于绘图。
- ImageGrab: 图片抓取模块。
- ImageOps: 图片处理模块。
- ImagePath: 路径队列模块。
- ImagePalette: 图片调色板模块。
- ImageSequence: 队列包装模块。
- ImageStat: 图片属性模块。
- ImageTk: 提供对 Tkinter 的支持。
- ImageWin: 提供对 Windows 的支持。
- PSDraw: 提供对 PostScript 的支持。

对于简单的图片处理一般仅需使用 Image 模块。此处仅对 Image 模块中的主要函数进行简要的介绍, 其他模块中的函数可以参考 PIL 的帮助文档。

1. 打开图片

Image 模块中主要的函数是 open 函数，其用于打开图片。函数原型如下所示。

```
open(file, mode)
```

其参数含义如下。

- file: 要打开的图片文件。
- mode: 可选参数，打开文件的方式。

2. 复制图片

open 函数执行成功后返回一个 Image 对象。使用 Image 模块的 copy 方法可以复制图片，使用 crop 方法可以复制图片中的某一区域，其原型如下所示。

```
crop(box)
```

其参数含义如下。

- box: 一个由 4 个元素组成的元组，分别表示图片的左、上、右、下的位置。

3. 粘贴图片

使用 Image 对象的 paste 方法可以向图片中粘贴图片、图像，paste 方法有以下几种形式。

```
paste(image, box)
paste(color, box)
paste(image, box, mask)
paste(color, box, mask)
```

其参数含义分别如下。

- image: 被粘贴到图片中的图片对象。
- box: 所要粘贴的区域，同 crop 中的参数 box。
- color: 填充的颜色。
- mask: 指定填充颜色透明度。

4. 调整图片大小

使用 Image 对象的 resize 方法可以重新调整图片的大小。其原型如下所示。

```
resize(size, filter)
```

其参数含义如下。

- size: 图片调整后的大小，为由宽和高组成的元组。
- filter: 可选参数，可以是 NEAREST、BILINEAR、BICUBIC 或者 ANTIALIAS。

5. 旋转图片

使用 Image 对象的 rotate 方法可以旋转图片。其原型如下所示。

```
rotate(angle, filter, expand)
```

其参数含义如下。

- angle: 旋转的角度。

- filter: 可选参数, 同 resize 中的 filter 参数。
- expand: 可选参数, 如果为真, 则增大图片, 以容纳旋转后的图片, 否则保持图片尺寸。

6. 显示图片

使用 Image 对象的 show 方法可以显示图片, 在 Windows 下 Image 模块将调用 Windows 图片和传真查看器显示图片。使用 Image 对象的 save 方法可以保存图像。其原型如下所示。

```
save(outfile, format, options)
```

其参数含义如下。

- outfile: 所保存的文件名。
- format: 可选参数, 保存的文件格式。如果不给出, 将根据保存的文件名的扩展名进行存储。
- options: 其他的操作参数。

另外, Image 对象还有 size 属性, 其为由宽和高组成的元组。Image 对象的 format 属性为图片的格式。Image 对象的 mode 属性为图片的模式。

23.2 使用 PIL 处理图片

PIL 提供了非常实用的函数, 很多情况下仅需使用一两个 PIL 的函数或者方法就可以完成对图片的处理, 如调整图片大小、转换图片格式等。配合 Python 的灵活性, 使用 PIL 可以创建一些非常实用的图片处理脚本。

23.2.1 转换图片格式

使用 PIL 转换图片格式, 主要使用 PIL 的 Image 模块。首先使用 Image.open 函数打开文件, 然后将文件保存成所需要的格式即可。Image 可以根据文件的扩展名自动选择文件保存的格式, 因此不需要设置文件格式。如下所示的 pyImageConv.py 脚本使用 Image 模块进行批量图片文件格式转换。

```
# -*- coding:utf-8 -*-
# file: pyImageConv.py
#
import os                                     # 导入模块
import Image
import Tkinter
import tkFileDialog
import tkMessageBox
class Window:                                # 创建窗口
    def __init__(self):
```



```

self.root = root = Tkinter.Tk()                                # 创建组件
label = Tkinter.Label(root, text = '选择目录')
label.place(x = 5, y = 5)
self.entry = Tkinter.Entry(root)
self.entry.place(x=55, y = 5)
self.buttonBrowser = Tkinter.Button(root, text = '浏览',
    command = self.Browser)
self.buttonBrowser.place(x=200, y = 5)
frameF = Tkinter.Frame(root)
frameF.place(x = 5, y = 30)
frameT = Tkinter.Frame(root)
frameT.place(x = 100, y = 30)
self.fImage = Tkinter.StringVar()                             # 生成关联变量
self.tImage = Tkinter.StringVar()
self.status = Tkinter.StringVar()
self.fImage.set('.bmp')
self.tImage.set('.bmp')
labelFrom = Tkinter.Label(frameF, text = 'From')
labelFrom.pack(anchor='w')
labelTo = Tkinter.Label(frameT, text = 'To')
labelTo.pack(anchor='w')
frBmp = Tkinter.Radiobutton(frameF, variable = self.fImage,
    value = '.bmp', text = 'BMP' )
frBmp.pack(anchor='w')
frJpg = Tkinter.Radiobutton(frameF, variable = self.fImage,
    value = '.jpg', text = 'JPG' )
frJpg.pack(anchor='w')
frGif = Tkinter.Radiobutton(frameF, variable = self.fImage,
    value = '.gif', text = 'GIF' )
frGif.pack(anchor='w')
frPng = Tkinter.Radiobutton(frameF, variable = self.fImage,
    value = '.png', text = 'PNG' )
frPng.pack(anchor='w')
trBmp = Tkinter.Radiobutton(frameT, variable = self.tImage,
    value = '.bmp', text = 'BMP' )
trBmp.pack(anchor='w')
trJpg = Tkinter.Radiobutton(frameT, variable = self.tImage,
    value = '.jpg', text = 'JPG' )
trJpg.pack(anchor='w')
trGif = Tkinter.Radiobutton(frameT, variable = self.tImage,
    value = '.gif', text = 'GIF' )
trGif.pack(anchor='w')
trPng = Tkinter.Radiobutton(frameT, variable = self.tImage,
    value = '.png', text = 'PNG' )
trPng.pack(anchor='w')
self.buttonConv = Tkinter.Button(root, text = '转换',
    command = self.Conv)
self.buttonConv.place(x=100, y = 150)
self.labelStatus = Tkinter.Label(root, textvariable=self.status)

```

```

        self.labelStatus.place(x=50, y = 175)
def MainLoop(self):                                # 进入消息循环
    self.root.minsize(250,200)
    self.root.maxsize(250,200)
    self.root.mainloop()
def Browser(self):                                  # 浏览目录
    directory = tkFileDialog.askdirectory(title='Python')
    if directory:
        self.entry.delete(0, Tkinter.END)
        self.entry.insert(Tkinter.END, directory)
def Conv(self):                                     # 转换文件格式
    n = 0
    t = self.tImage.get()
    f = self.fImage.get()
    path = self.entry.get()
    if path == '':
        tkMessageBox.showerror('Python Tkinter','请输入路径')
        return
    filenames = os.listdir(path)
    os.mkdir(path + '/' + t[-3:])
    for filename in filenames:
        if filename[-4:] == f:
            Image.open(path + '/' + filename).save(path +
            '/' + t[-3:] + '/' + filename[:-4] + t)
        n = n + 1
    self.status.set('成功转换%d张图片' % n)
window = Window()
window.MainLoop()

```

运行 pyImageConv.py 脚本后如图 23-4 所示。

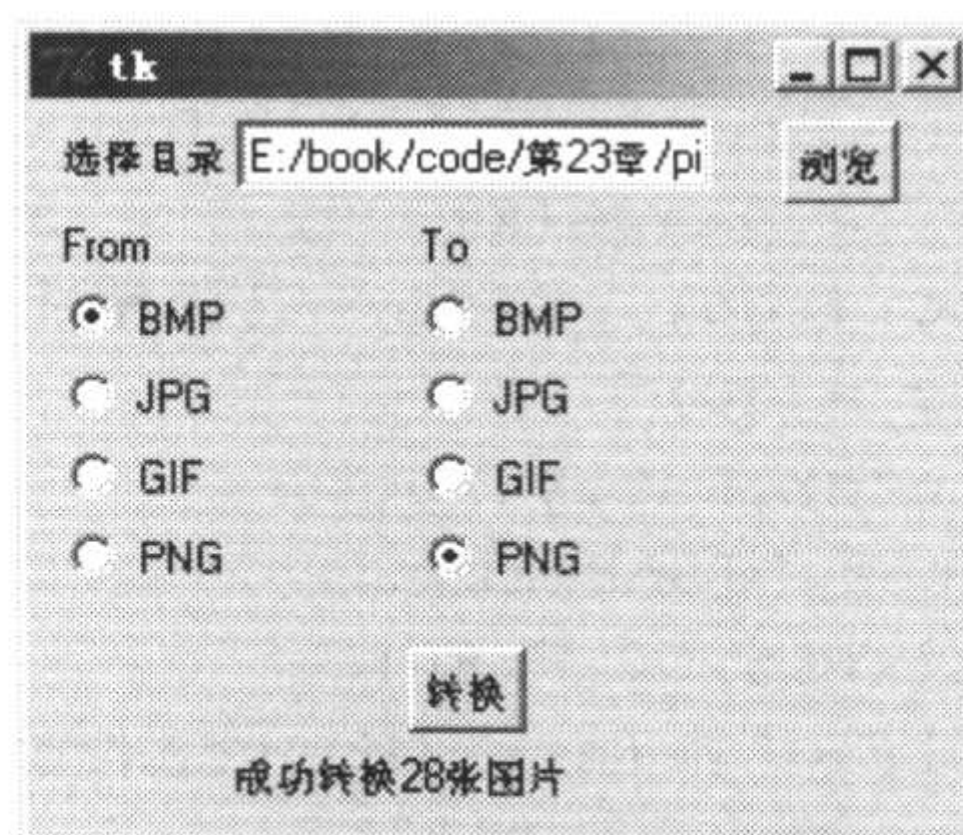


图 23-4 使用 PIL 转换文件格式

23.2.2 生成缩略图

使用 PIL 生成缩略图主要使用 Image 的 resize 函数。使用 resize 函数可以重新指定图片的大小。如下所示的 pyImageThumb.py 脚本使用 PIL 模块批量生成图片缩略图。

```

# -*- coding:utf-8 -*-
# file: pyImageThumb.py

```



```
#
import os                                     # 导入模块
import Image
import Tkinter
import tkFileDialog
import tkMessageBox
class Window:
    def __init__(self):
        self.root = root = Tkinter.Tk()
        self.Image = Tkinter.StringVar()
        self.status = Tkinter.StringVar()
        self.mstatus = Tkinter.IntVar()
        self.fstatus = Tkinter.IntVar()
        self.Image.set('bmp')
        self.mstatus.set(0)
        self.fstatus.set(0)
        label = Tkinter.Label(root, text = '输入百分比')
        label.place(x = 5, y = 5)
        self.entryNew = Tkinter.Entry(root)
        self.entryNew.place(x = 65, y = 5)
        self.checkM = Tkinter.Checkbutton(root, text = '批量转换',
            command = self.OnCheckM,
            variable = self.mstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkM.place(x = 5, y = 30)
        label = Tkinter.Label(root, text = '选择文件')
        label.place(x = 5, y = 55)
        self.entryFile = Tkinter.Entry(root)
        self.entryFile.place(x=55, y = 55)
        self.buttonBrowserFile = Tkinter.Button(root, text = '浏览',
            command = self.BrowserFile)
        self.buttonBrowserFile.place(x=200, y = 55)
        label = Tkinter.Label(root, text = '选择目录')
        label.place(x = 5, y = 80)
        self.entryDir = Tkinter.Entry(root,
            state = Tkinter.DISABLED)
        self.entryDir.place(x=55, y = 80)
        self.buttonBrowserDir = Tkinter.Button(root, text = '浏览',
            command = self.BrowserDir,
            state = Tkinter.DISABLED)
        self.buttonBrowserDir.place(x=200, y = 80)
        self.checkF = Tkinter.Checkbutton(root, text = '改变文件格式',
            command = self.OnCheckF,
            variable = self.fstatus,
            onvalue = 1,
            offvalue = 0)
        self.checkF.place(x = 5, y = 110)
        frame = Tkinter.Frame(root)
```

```

frame.place(x = 10, y = 130)
labelTo = Tkinter.Label(frame, text = 'To')
labelTo.pack(anchor='w')
self.rBmp = Tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'bmp', text = 'BMP',
                                state = Tkinter.DISABLED)
self.rBmp.pack(anchor='w')
self.rJpg = Tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'jpg', text = 'JPG',
                                state = Tkinter.DISABLED)
self.rJpg.pack(anchor='w')
self.rGif = Tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'gif', text = 'GIF',
                                state = Tkinter.DISABLED)
self.rGif.pack(anchor='w')
self.rPng = Tkinter.Radiobutton(frame, variable = self.Image,
                                value = 'png', text = 'PNG',
                                state = Tkinter.DISABLED)
self.rPng.pack(anchor='w')
self.buttonConv = Tkinter.Button(root, text = '转换',
                                command = self.Conv)
self.buttonConv.place(x=100, y = 175)
self.labelStatus = Tkinter.Label(root, textvariable=self.status)
self.labelStatus.place(x=80, y = 205)
def MainLoop(self):                                     # 进入消息循环
    self.root.minsize(250,250)
    self.root.maxsize(250,250)
    self.root.mainloop()
def BrowserDir(self):                                   # 选择路径
    directory = tkFileDialog.askdirectory(title='Python')
    if directory:
        self.entryDir.delete(0, Tkinter.END)
        self.entryDir.insert(Tkinter.END, directory)
def BrowserFile(self):                                  # 选择文件
    file = tkFileDialog.askopenfilename(title = 'Python Music Player',
        filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
        ('GIF', '*.gif'), ('PNG', '*.png')])
    if file:
        self.entryFile.delete(0, Tkinter.END)
        self.entryFile.insert(Tkinter.END, file)
def OnCheckM(self):                                     # 设置组件状态
    if not self.mstatus.get():
        self.entryDir.config(state = Tkinter.DISABLED)
        self.entryFile.config(state = Tkinter.NORMAL)
        self.buttonBrowserDir.config(state = Tkinter.DISABLED)
        self.buttonBrowserFile.config(state = Tkinter.NORMAL)
    else:
        self.entryDir.config(state = Tkinter.NORMAL)
        self.entryFile.config(state = Tkinter.DISABLED)

```



```

        self.buttonBrowserDir.config(state = Tkinter.NORMAL)
        self.buttonBrowserFile.config(state = Tkinter.DISABLED)
def OnCheckF(self):
    if not self.fstatus.get():
        self.rBmp.config(state = Tkinter.DISABLED)
        self.rJpg.config(state = Tkinter.DISABLED)
        self.rGif.config(state = Tkinter.DISABLED)
        self.rPng.config(state = Tkinter.DISABLED)
    else:
        self.rBmp.config(state = Tkinter.NORMAL)
        self.rJpg.config(state = Tkinter.NORMAL)
        self.rGif.config(state = Tkinter.NORMAL)
        self.rPng.config(state = Tkinter.NORMAL)
def Conv(self):
    n = 0
    if self.mstatus.get():
        path = self.entryDir.get()
        if path == '':
            tkMessageBox.showerror('Python Tkinter', '请输入路径')
            return
        filenames = os.listdir(path)
        if self.fstatus.get():
            f = self.Image.get()
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                    self.make(path + '/' + filename, f)
                    n = n + 1
        else:
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                    self.make(path + '/' + filename)
                    n = n + 1
    else:
        file = self.entryFile.get()
        if file == '':
            tkMessageBox.showerror('Python Tkinter', '请选择文件')
            return
        if self.fstatus.get():
            f = self.Image.get()
            self.make(file, f)
            n = n + 1
        else:
            self.make(file)
            n = n + 1
    self.status.set('成功转换%d 图片' % n)
def make(self, file, format = None):
    im = Image.open(file)
    mode = im.mode
    if mode not in ('L', 'RGB'):

```

设置组件状态

转换图片

生成缩略图

```

        im = im.convert('RGB')
width, height = im.size
s = self.entryNew.get()
if s == '':
    tkMessageBox.showerror('Python Tkinter', '请输入百分比')
    return
else:
    n = int(s)
    nwidth = width * n / 100
    nheight = height * n / 100
    thumb = im.resize((nwidth, nheight), Image.ANTIALIAS)
    if format:
        thumb.save(file[: (len(file) - 4)] + '_thumb.' + format)
    else:
        thumb.save(file[: (len(file) - 4)] + '_thumb' + file[-4:])
window = Window()
window.MainLoop()

```

运行 pyImageThumb.py 后的效果如图 23-5 所示。

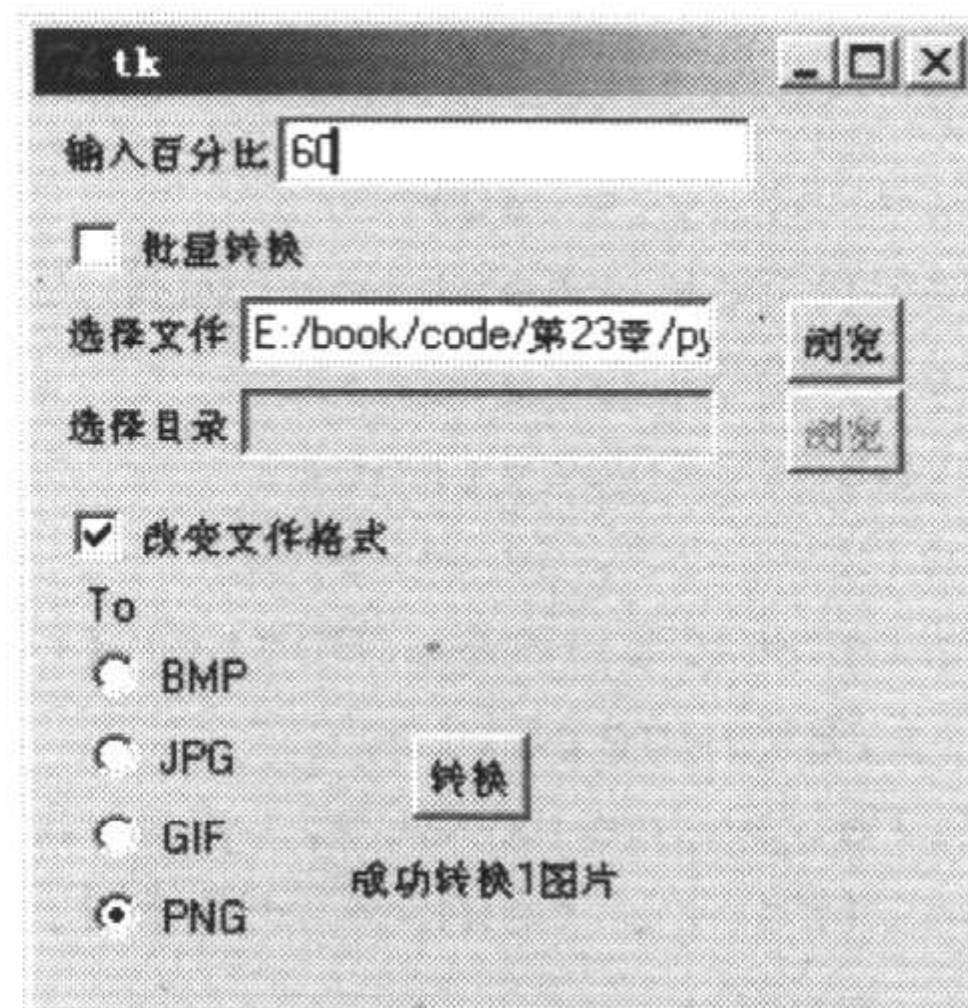


图 23-5 使用 PIL 生成缩略图

23.2.3 为图片添加 Logo

使用 PIL 为图片添加 Logo，主要使用 Image 的 paste 函数。paste 函数可以向图片中粘贴其他的图片。如下所示的 pyImageAddLogo.py 脚本使用 PIL 模块为图片批量添加 Logo。

```

# -*- coding:utf-8 -*-
# file: pyImageAddLogo.py
#
import os
import Image
import Tkinter
import tkFileDialog
import tkMessageBox
class Window:
    def __init__(self):
        self.root = root = Tkinter.Tk()

```

导入模块

创建组件


```
self.Image = Tkinter.StringVar()
self.status = Tkinter.StringVar()
self.mstatus = Tkinter.IntVar()
self.fstatus = Tkinter.IntVar()
self.pstatus = Tkinter.IntVar()
self.Image.set('bmp')
self.mstatus.set(0)
self.fstatus.set(0)
self.pstatus.set(0)
label = Tkinter.Label(root, text = 'Logo')
label.place(x = 5, y = 5)
self.entryLogo = Tkinter.Entry(root)
self.entryLogo.place(x = 55, y = 5)
self.buttonBrowserLogo = Tkinter.Button(root, text = '浏览',
    command = self.BrowserLogo)
self.buttonBrowserLogo.place(x = 200, y = 5)
self.checkM = Tkinter.Checkbutton(root, text = '批量转换',
    command = self.OnCheckM,
    variable = self.mstatus,
    onvalue = 1,
    offvalue = 0)
self.checkM.place(x = 5, y = 30)
label = Tkinter.Label(root, text = '选择文件')
label.place(x = 5, y = 55)
self.entryFile = Tkinter.Entry(root)
self.entryFile.place(x = 55, y = 55)
self.buttonBrowserFile = Tkinter.Button(root, text = '浏览',
    command = self.BrowserFile)
self.buttonBrowserFile.place(x=200, y = 55)
label = Tkinter.Label(root, text = '选择目录')
label.place(x = 5, y = 80)
self.entryDir = Tkinter.Entry(root,
    state = Tkinter.DISABLED)
self.entryDir.place(x=55, y = 80)
self.buttonBrowserDir = Tkinter.Button(root, text = '浏览',
    command = self.BrowserDir,
    state = Tkinter.DISABLED)
self.buttonBrowserDir.place(x=200, y = 80)
self.checkF = Tkinter.Checkbutton(root, text = '改变文件格式',
    command = self.OnCheckF,
    variable = self.fstatus,
    onvalue = 1,
    offvalue = 0)
self.checkF.place(x = 5, y = 110)
frame = Tkinter.Frame(root)
frame.place(x = 10, y = 130)
labelTo = Tkinter.Label(frame, text = '格式')
labelTo.pack(anchor='w')
self.rBmp = Tkinter.Radiobutton(frame, variable = self.Image,
```

第23章 使用PIL处理图片

```

        value = 'bmp', text = 'BMP',
        state = Tkinter.DISABLED)
self.rBmp.pack(anchor='w')
self.rJpg = Tkinter.Radiobutton(frame, variable = self.Image,
    value = 'jpg', text = 'JPG',
    state = Tkinter.DISABLED)
self.rJpg.pack(anchor='w')
self.rGif = Tkinter.Radiobutton(frame, variable = self.Image,
    value = 'gif', text = 'GIF',
    state = Tkinter.DISABLED)
self.rGif.pack(anchor='w')
self.rPng = Tkinter.Radiobutton(frame, variable = self.Image,
    value = 'png', text = 'PNG',
    state = Tkinter.DISABLED)
self.rPng.pack(anchor='w')
pframe = Tkinter.Frame(root)
pframe.place(x = 70, y = 130)
labelPos = Tkinter.Label(pframe, text = '位置')
labelPos.pack(anchor = 'w')
self.rLT = Tkinter.Radiobutton(pframe, variable = self.pstatus,
    value = 0, text = '左上角')
self.rLT.pack(anchor = 'w')
self.rRT = Tkinter.Radiobutton(pframe, variable = self.pstatus,
    value = 1, text = '右上角')
self.rRT.pack(anchor = 'w')
self.rLB = Tkinter.Radiobutton(pframe, variable = self.pstatus,
    value = 2, text = '左下角')
self.rLB.pack(anchor = 'w')
self.rRB = Tkinter.Radiobutton(pframe, variable = self.pstatus,
    value = 3, text = '右下角')
self.rRB.pack(anchor = 'w')
self.buttonAdd = Tkinter.Button(root, text = '添加',
    command = self.Add)
self.buttonAdd.place(x=180, y = 175)
self.labelStatus = Tkinter.Label(root, textvariable=self.status)
self.labelStatus.place(x=150, y = 205)
def MainLoop(self):
    self.root.minsize(250,250)
    self.root.maxsize(250,250)
    self.root.mainloop()
def BrowserLogo(self):
    file = tkFileDialog.askopenfilename(title = 'Python:Music Player',
        filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
            ('GIF', '*.gif'), ('PNG', '*.png')])
    if file:
        self.entryLogo.delete(0, Tkinter.END)
        self.entryLogo.insert(Tkinter.END, file)
def BrowserDir(self):
    directory = tkFileDialog.askdirectory(title='Python')

```

进入消息循环

选择路径


```

        if directory:
            self.entryDir.delete(0, Tkinter.END)
            self.entryDir.insert(Tkinter.END, directory)
def BrowserFile(self):                                     # 选择文件
    file = tkFileDialog.askopenfilename(title = 'Python Music Player',
        filetypes=[('JPG', '*.jpg'), ('BMP', '*.bmp'),
            ('GIF', '*.gif'), ('PNG', '*.png')])
    if file:
        self.entryFile.delete(0, Tkinter.END)
        self.entryFile.insert(Tkinter.END, file)
def OnCheckM(self):                                       # 设置组件状态
    if not self.mstatus.get():
        self.entryDir.config(state = Tkinter.DISABLED)
        self.entryFile.config(state = Tkinter.NORMAL)
        self.buttonBrowserDir.config(state = Tkinter.DISABLED)
        self.buttonBrowserFile.config(state = Tkinter.NORMAL)
    else:
        self.entryDir.config(state = Tkinter.NORMAL)
        self.entryFile.config(state = Tkinter.DISABLED)
        self.buttonBrowserDir.config(state = Tkinter.NORMAL)
        self.buttonBrowserFile.config(state = Tkinter.DISABLED)
def OnCheckF(self):                                       # 设置组件状态
    if not self.fstatus.get():
        self.rBmp.config(state = Tkinter.DISABLED)
        self.rJpg.config(state = Tkinter.DISABLED)
        self.rGif.config(state = Tkinter.DISABLED)
        self.rPng.config(state = Tkinter.DISABLED)
    else:
        self.rBmp.config(state = Tkinter.NORMAL)
        self.rJpg.config(state = Tkinter.NORMAL)
        self.rGif.config(state = Tkinter.NORMAL)
        self.rPng.config(state = Tkinter.NORMAL)
def Add(self):                                           # 处理图片
    n = 0
    if self.mstatus.get():
        path = self.entryDir.get()
        if path == '':
            tkMessageBox.showerror('Python Tkinter', '请输入路径')
            return
        filenames = os.listdir(path)
        if self.fstatus.get():
            f = self.Image.get()
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):
                    self.addlogo(path + '/' + filename, f)
                    n = n + 1
        else:
            for filename in filenames:
                if filename[-3:] in ('bmp', 'jpg', 'gif', 'png'):

```

```

        self.addlogo(path + '/' + filename)
        n = n + 1
    else:
        file = self.entryFile.get()
        if file == '':
            tkinterMessageBox.showerror('Python Tkinter', '请选择文件')
            return
        if self.fstatus.get():
            f = self.Image.get()
            self.addlogo(file, f)
            n = n + 1
        else:
            self.addlogo(file)
            n = n + 1
    self.status.set('成功添加%d张图片' % n)
def addlogo(self, file, format = None):
    logo = self.entryLogo.get()
    if logo == '':
        tkinterMessageBox.showerror('Python Tkinter', '请选择 Logo')
        return
    im = Image.open(file)
    lo = Image.open(logo)
    imwidth = im.size[0]
    imheight = im.size[1]
    lowidth = lo.size[0]
    loheight = lo.size[1]
    pos = self.pstatus.get()
    if pos == 0:
        left = 0
        top = 0
        right = lowidth
        bottom = loheight
    elif pos == 1:
        left = imwidth - lowidth
        top = 0
        right = imwidth
        bottom = loheight
    elif pos == 2:
        left = 0
        top = imheight - loheight
        right = lowidth
        bottom = imheight
    else:
        left = imwidth - lowidth
        top = imheight - loheight
        right = imwidth
        bottom = imheight
    im.paste(lo, (left, top, right, bottom))
    if format:

```

向图片添加 Logo


```

        im.save(file[: (len(file) - 4)] + '_logo.' + format)
    else:
        im.save(file[: (len(file) - 4)] + '_logo' + file[-4:])
window = Window()
window.MainLoop()

```

运行 pyImageAddLogo.py 脚本后的效果如图 23-6 所示。

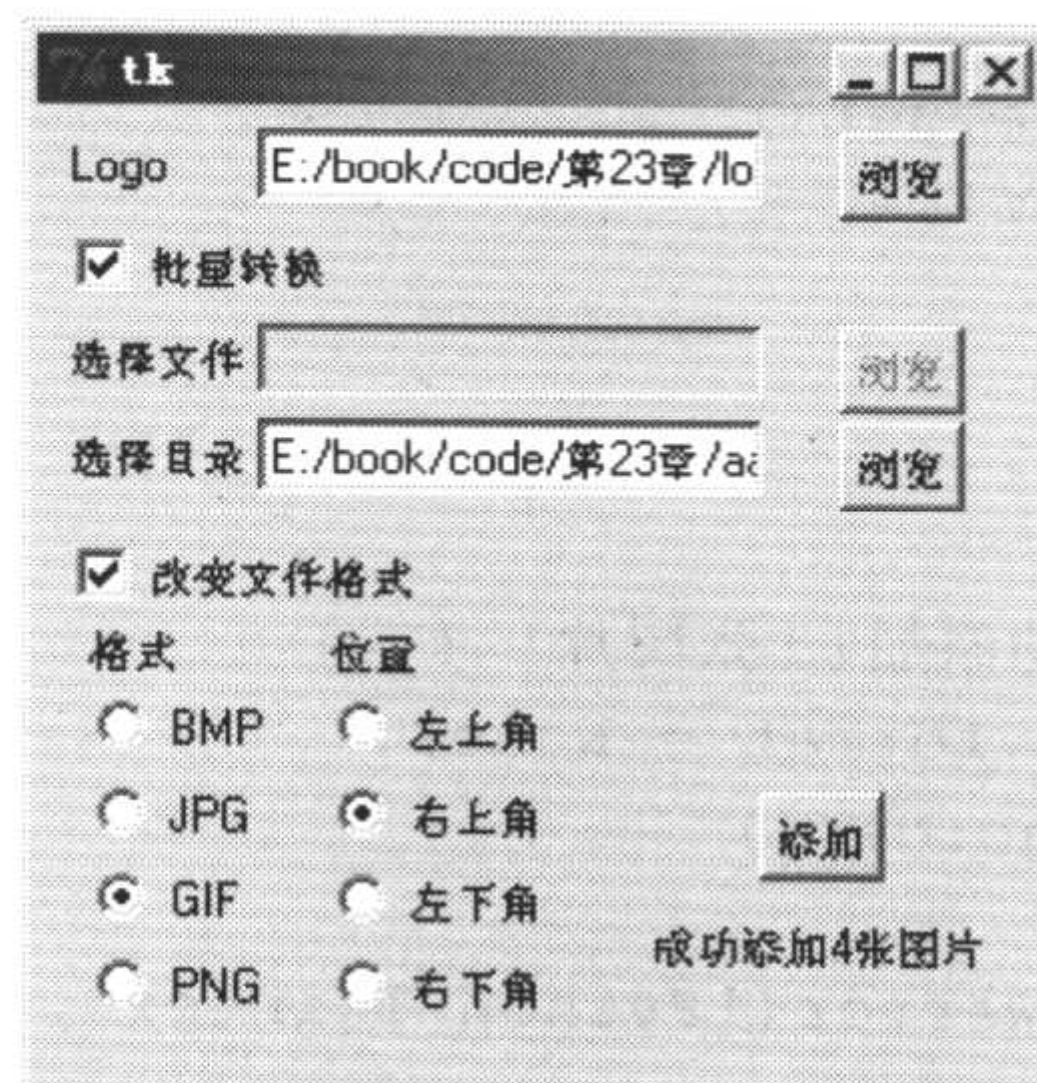


图 23-6 使用 PIL 为图片添加 Logo

[G e n e r a l I n f o r m a t i o n]
书名 = 征服 P Y T H O N : 语言基础与典型应用
作者 = 孙广磊编著
页数 = 4 7 6
S S 号 = 1 1 8 6 9 4 2 3
出版日期 = 2 0 0 7 年 0 9 月 第 1 版
出版社 = 人民邮电出版社

[G e n e r a l I n f o r m a t i o n]
书名 = 征服 P Y T H O N : 语言基础与典型应用
作者 = 孙广磊编著
页数 = 4 7 6
S S 号 = 1 1 8 6 9 4 2 3
出版日期 = 2 0 0 7 年 0 9 月 第 1 版
出版社 = 人民邮电出版社